Math for Programmers

Tobias Schoofs

July 22, 2017

Contents

1. Introduction **9**

2. Brief Introduction to Haskell **11**

I. Arithmetic and Combinatorics **13**

3. Natural Numbers **15**
- 3.1. What are numbers and what should they be? 15
- 3.2. Peano Numbers 17
- 3.3. Decimal Numbers 19
- 3.4. Multiplication 30
- 3.5. Division and the Greatest Common Divisor 35
- 3.6. Powers, Roots and Logarithms 41
- 3.7. Numbers as Strings 44
- 3.8. $\mathbb{N}$ 50
- 3.9. Abstract Algebra 52

4. Induction, Series, Sets and Combinatorics **57**
- 4.1. Logistics 57
- 4.2. Induction 61
- 4.3. Arithmetic and Geometric Series 65
- 4.4. The Fibonacci Sequence 65
- 4.5. Factorial 68
- 4.6. Random Permutations 74
- 4.7. Binomial Coefficients 79
- 4.8. Combinatorial Problems with Sets 88
- 4.9. Stirling Numbers 98
- 4.10. Eulerian Numbers 111

5. Primes **119**
- 5.1. Gaps 119
- 5.2. Finding Primes 121
- 5.3. The fundamental Theorem of Arithmetic 126
- 5.4. Factoring 130
- 5.5. Factoring Factorials 135
- 5.6. Arithmetic modulo a Prime 137

5.7. Congruence 149
5.8. Quadratic Residues 155
5.9. Generators and Subgroups 167
5.10. Primality Tests 174
5.11. Primes in Cryptography 183
5.12. Open Problems 194

6. The Inverse Element 203
6.1. Inverse Elements 203
6.2. $\mathbb{Z}$ 207
6.3. Negative Binomial Coefficients 211
6.4. $\mathbb{Q}$ 224
6.5. Zeno's Paradox 227
6.6. Systems of Linear Equations 232
6.7. Binomial Coefficients are Integers 245
6.8. Euler's Totient Function 256
6.9. How many Fractions? 271

7. Real Numbers 287
7.1. $\sqrt{2}$ 287
7.2. Φ 289
7.3. π 291
7.4. e 295
7.5. γ 300
7.6. Representation of Real Numbers 304
7.7. $\mathbb{R}$ 306
7.8. Real Factorials 310
7.9. The Stern-Brocot Tree 317
7.10. Field Extension 326
7.11. The Continuum 336
7.12. Review of the Number Zoo 340

II. Algebra and Geometry 343

8. Polynomials 345
8.1. Numeral Systems 345
8.2. Polynomial Arithmetic 351
8.3. The Difference Engine 362
8.4. Differences and Binomial Coefficients 372
8.5. Umbral Calculus 381
8.6. Roots 381
8.7. Vieta's Formula 388
8.8. Factoring Polynomials 388

8.9. Factoring Polynomials in a finite Field 392
8.10. Hensel Lifting 406
8.11. The closed Form of the Fibonacci Sequence 406

9. Relations, Functions and the Cartesian Plane 409
9.1. Relations 409
9.2. Equivalence Classes 409
9.3. Functions 409
9.4. Plotting Functions 409
9.5. Slopes and Intersects 409
9.6. Trigonometry 409
9.7. Navigation 409
9.8. Conic Sections 409
9.9. Algebra and Geometry 409

10. Linear Algebra 411
10.1. Vectors 411
10.2. Vector Spaces 411
10.3. Clustering 411
10.4. Linear Maps and Operators 411
10.5. The Matrix 411
10.6. Eigenvalues 411
10.7. Inner Products and their Operators 411
10.8. Polynomials and Vector Spaces 411
10.9. Determinants 411
10.10Support Vector Machines 411

11. Elliptic Curves 413
11.1. Geometry Intuition 413
11.2. Projective Geometry 420
11.3. EC modulo a Prime 425
11.4. The EC Discrete Logarithm Problem 432
11.5. EC Diffie-Hellman 436
11.6. EC Integrated Encryption Scheme 441
11.7. EC Digital Signature Algorithm 444
11.8. Cryptoanalysis 447
11.9. Mr. Frobenius 447
11.10Mr. Schoof 447
11.11Mr. Elkies and Mr. Aktin 447
11.12EC in Practice 447

12. Complex Numbers 449
12.1. i 449
12.2. Complex Numbers 449

12.3. The complex Plane . . . 449
12.4. Polar Form . . . 449
12.5. $\mathbb{C}$ . . . 449
12.6. Gaussian Integers . . . 449
12.7. $\mathbb{Q}(i)$ . . . 449
12.8. 449
12.9. Complex Vector Spaces . . . 449
12.10... . . . 449
12.11Quantum Mechanics . . . 449
12.12Hypercomplex Numbers . . . 449

13. Quadratics, Cubics and Quartics 451
13.1. Quadratic Equations . . . 451
13.2. The fundamental Theorem of Algebra . . . 451
13.3. Cubics and Quartics . . . 451
13.4. Quintics . . . 451
13.5. Galois Theory . . . 451
13.6. Field Theory . . . 451
13.7. Postmodern Algebra . . . 451
13.8. Category Theory . . . 451

14. Euclid and beyond 453
14.1. Euclid's Geometry . . . 453
14.2. The Parallel Postulate . . . 453
14.3. Non-Euclidian Geometries . . . 453
14.4. Lost in Hilbert Space . . . 453
14.5. Geometry and Relativity . . . 453
14.6. Affine Geometry . . . 453
14.7. Projective Geometry . . . 453
14.8. Graph Theory . . . 453
14.9. Topology . . . 453

III. Analysis 455

15. The Calculus 457
15.1. The Method of Exhaustion . . . 457
15.2. Area below a Curve . . . 457
15.3. Basic Rules of Integration . . . 457
15.4. Minima, Maxima and Slopes . . . 457
15.5. Basic Rules of Derivation . . . 457
15.6. The fundamental Theorem of the Calculus . . . 457
15.7. More Rules . . . 457

16. Curve Sketching **459**

17. Advanced Function Analysis **461**

18. Generating Functions **463**

19. Applied Mathematics **465**

20. Prospects **467**

1. Introduction

2. Brief Introduction to Haskell

Part I.

Arithmetic and Combinatorics

3. Natural Numbers

3.1. What are numbers and what should they be?

In his famous article “What are numbers and what should they be?” German mathematician Richard Dedekind (1831 – 1916) described numbers as “free creations of the human mind”. This idea is anything but self-evident. Dedekind’s contemporary and fellow combatant for a new approach in math, namely set theory, Georg Cantor fiercely defended *mathematical platonism*, the idea that mathematical objects are real existing and that mathematical methods are tools that allow us to see these objects like, perhaps, telescopes, enable us to examine stars far away in the universe.

This view clashed with the view of the *constructivist* Leopold Kronecker, a resolute opponent to the novelties introduced by Cantor, Dedekind and others. In the view of the constructivists like Kronecker only those objects that can be constructed in a finite number of steps from a finite number of integers are valid objects of mathematics. Anything else is a chimaera. “The natural numbers are made by god”, as he put it, “all else is the work of man.”

Both views, constructivism and platonism, even though not mainlines of discussion anymore, are still influential today. The influence of platonism is visible, when it is said that some mathematician has “discovered” a concept, *e.g.* “Gauss discovered a construction of the heptadecagon”. Mathematicians in the tradition of constructivism or its modern variant *intuitivism*, insist in a terminology that stresses that mathematical concepts are man-made. They would say Gauss has “invented”, “developed” or, simply, “constructed”.

It is consensus, however, that, independent of the question whether numbers are real existing things or invented concepts, there must be some kind of representation of numbers in our mind. It is also likely that these representations have some grounding in the nervous system and are thus part of our genetic legacy. A strong evidence therefor is the fact that some animals can distinguish sets of different numbers of objects. Birds, for instance, realise when an egg is missing in their nest, but do not necessarily realise that an egg was replaced by another one. That would suggest that the “creation” of numbers is not as “free” as it appears to be according to Dedekind. The ability to work with numbers would then be part of our biology, just like having language capability is part of our biology.

When we try to reduce numbers to the very heart of what they are, we quickly come to the notion of counting. The assumption that numbers basically represent steps in the process of counting is supported by the fact that many number systems in use during history are modelled on something countable like fingers (leading to the decimal number system), fingers and toes (leading to the vigesimal system), finger bones (leading to systems based on multiples of 12) and so on. According to this view, relations between quantities (length, volume, weight, *etc.*) would represent more abstract concepts engineered in some way on top of the fundamental notion of counting. It is not so clear, however, if this hierarchy corresponds to the real historic development. Among the oldest known math problems are such as determining the quanitity of corn for baking bread in acient Egypt or investigating rhythmic patterns formed by single and multiple beats. Problems of this kind cannot be solved by mere counting, but demand comparison of *continuous* quantities like volume, weight or time intervals leading immediately to fractions. Numbers may hence be related to a human sense of quantity that is more general than counting.

Numbers appear to be very simple and clear concepts. But, in fact, together with the operations defined on them, addition, multiplication, subtraction and division, they quickly give rise to intricate problems. Those problems are not like add-ons to the numbers that were invented later to complicate things, they belong to the very core of the number concept. It appears, in fact, that those problems have already slumbered in the apparently simple concept, but needed some time to unfold themselves – centuries in some cases, millennia in others. Even if the number concept may have been created by the human mind, once fixed, all its consequences and complications are there. In this sense, every mathematician would agree to speak of “discovering” a problem in a certain field of math. Mathematicians may still disagree, however, on whether the solution to such a problem was “discovered” or “developed”.

The most notorious group of long-standing problems is number theory, which deals mainly with questions of prime numbers, *i.e.* natural numbers that cannot be constructed from other natural numbers using multiplication. Prime numbers are investigated for some three thousand years now, but, still, many fundamental questions remain unanswered. Infamous are the number-theoretic theorems drafted, but not proven, by Pierre de Fermat in the 17^{th} century. Most of these theorems look quite innocent, but by proving them in the course of the following three centuries, math made huge progress developing methods that had been unknown to Fermat and his contemporaries.

The most insistent problem, known as *Fermat’s Last Theorem*, was solved only in the late 20^{th} century by British mathematician Andrew Wiles. What makes this specific problem so mind-boggling is its extremely simple formulation – compared to the proof of some 200 pages. Indeed, Fermat doodled the theorem on the margin of a page in his copy of an acient math book, Diophantos’ “Arithmetica”, and, to the irritation of generations of mathematicians, he added: “I have found a really wonderful proof, but, unfortunately, this margin is too narrow to contain it.” The theorem in question states that there is no solution for equations of the form: $a^z + b^z = c^z$ for a, b, c and z all natural numbers and $z > 2$. For $z = 2$, many solutions exist, the so called *Pythagorean triples*, *e.g.*: $3^2 + 4^2 = 5^2$.

Another set of difficult problems is algebra and the search for solutions of higher-degree equations, which caused a lot of headache throughout the centuries. The general solution for quadratic equations of the form $ax^2+bx+c=0$ was known since antiquity. Solutions for cubic and quartic equations were found by joint efforts and fierce competition in the 16^{th} century by Italian mathematicians, namely Niccolò Fontana Tartaglia, Gerolamo Cardano and Lodovico Ferrari. But it took until the early 19^{th} century, before mathematicians were able to make statements about quintic and even higher-degree equations. In fact, the investigations took an unpredicted direction and led to the development of a new branch of mathematics, *group theory*, which studies sets of objects – not only numbers – and operations defined on these objects, and *abstract algebra*, which generalises group theory making objects of completely different areas of mathematics comparable.

Group theory, which was developed during the 18^{th} and 19^{th} century by Joseph-Louis Lagrange, Paolo Ruffini, young Niels Henrik Abel and the incredible Évariste Galois, is a great helper in organising the number zoo that developed out of simpler notions of numbers in the course of the time. At the beginning – if we believe in numbers being fundamentally related to counting – there may have been only the concept of natural numbers, which was then extended by adding fractions to cope with relations between quantities and negative numbers to deal with negative quantities like debts. The investigations into geometry by Greek mathematicians led to the rise of irrational numbers – such as the number π – which, as a surprise to acient mathematicians, cannot be expressed in terms of ratios of integers. The studies in algebra led to complex numbers, which, in their turn, inspired the construction of hypercomplex numbers.

In this chapter, we will start with the simplest possible number concept, natural numbers, and we will stick to it as long as possible – perhaps longer. We then will make a big leap introducing negative numbers and fractions. Afterwards we enter the confusing world of irrational numbers and will probably understand the annoyance of mathematicians like Kronecker when faced with mathematical concepts dealing with these strange creatures. Later, complex numbers and even more exotic beasts will come into focus. On the way, we will introduce some basic group theory helping us to find our way in the number jungle.

3.2. Peano Numbers

The Italian mathematician Giuseppe Peano (1858 – 1932) defined an axiomatic system to describe the natural numbers and their properties. An axiomatic system consists of axioms, statements that are known or simply assumed to be true, and rules that describe how other statements can be derived from axioms, such that these new statements are true if the axioms are true. In practice the procedure is usually followed the other way round: one would try to find a sequence of applications of the rules that links a given statement with one or more axioms. This process is called a *proof* and is one of the main things mathematicians do to kill their time.

A major part of the discussions about the foundations of math in the first half of the 20th century was about the idea formulated by David Hilbert to construct the whole of mathematics as an axiomatic system and, then, to prove that for every statement that is believed to be true a sequence of rule applications can be found that derives this statement from the axioms. The plan failed in the 1930ies, after releasing an incredible amount of mathematical creativity that resulted, among other things, in a theoretical model of a universal computing device, known today as the *Turing machine* and in the *lambda calculus*, which, as already discussed, is one of the foundations of functional programming. The first task of the Turing machine and of the lambda calculus was indeed to prove that Hilbert's plan is impossible.

Peano's objective was in Hilbert's line: to provide a foundation of natural numbers as a first step towards an axiomatisation of the whole field of arithmetic. In spite of this ambitious goal, Peano's axioms are quite simple. The basic idea is to define natural numbers by two elements: The explicitly defined number *Zero* and a recursive function *Successor* that defines any other number. Peano's axioms boil down to a formulation in Haskell like:

```
data Peano = Zero | S Peano
  deriving Show
```

This captures very well the process of counting. Instead of adding 1 to a given number, we just derive its successor:

```
succ :: Peano → Peano
succ p = S p
```

For instance, the successor of *Zero*, one, is: $S(Zero)$; two is $S(S(Zero))$, three is $S(S(S(Zero)))$ and so on.

We can also define a function to count backwards, *i.e.*:

```
pre :: Peano → Peano
pre Zero  = ⊥
pre (S p) = p
```

$Zero$, this is one of Peano's axioms, has no predecessor. The predecessor of any other number, is that number with one S removed. The predecessor of $S(S(S(S(S(Zero)))))$, five, for instance, is $S(S(S(S(Zero))))$, four.

We can also devise a simple addition function:

```
add :: Peano → Peano → Peano
add Zero a  = a
add a Zero  = a
add (S a) b = add a (S b)
```

Subtraction is implemented easily as well:

```
sub :: Peano → Peano → Peano
sub a Zero        = a
sub Zero a        = ⊥
sub (S a) (S b) = sub a b
```

Note that any number reduced by *Zero* is just that number. *Zero* reduced by any number but *Zero*, however, is undefined. For all other cases, we just reduce a and b by one S until we hit one of the base cases.

We could go on and define multiplication, division and all other arithmetic operations based on this notion of numbers. It is indeed convincing in its simplicity and used as a standard system for research into arithmetic until today. But it is not very convenient, especially when working with huge numbers. Peano numbers are unary numbers, that is, to represent the number 100, one has to write 100 symbols (in fact, 101 symbols: 100 S and 1 *Zero*). As a number system, Peano numbers are even less handy than the roman numerals, which introduce symbols at least for some greater values, such as V, X, L and C. A trick that is often used in literature to mitigate this shortcoming is to add a subscript number to the S symbol to make clear, how many Ses we would have to write to represent this value, for instance, $S_5(Zero)$ would be 5. But, of course, that makes the use of Peano numbers – as a number system – pointless. Already Peano was faced with the clumsiness of his axioms when used as a number system: he tried to use it as a didactic device in his teaching both at the university and at the military academy where he was working. In the case of the military academy, this led to desaster and, eventually, to his dismissal in 1901. His achievements as mathetmatician and logician, however, were respected in the scientific community worldwide.

Let us learn from Peano's didactic failure and look out for a more practical number system, one that allows us to use significantly fewer symbols than the value of the number we want to represent. A system that *scales* in this sense is our well-known decimal number system.

3.3. Decimal Numbers

A numeral system consists of a vocabulary of symbols, which we will call *digits*, rules that define how to compose digits to strings and a model that leads to an arithmetic interpretation of such strings. To make practical use of the numeral system, we must also define a set of basic operations, such as counting forward and backward, addition, subtraction, multiplication, division and whatever we want to do with our numbers.

We define the following vocabulary:

```
data Digit = Zero | One | Two   | Three | Four |
             Five | Six | Seven | Eight | Nine
  deriving (Show, Eq, Ord)
```

Numbers are lists of *Digit*s:

```
type Number = [Digit]
```

Some numbers that are used more often than all others are:

```
zero, unity, two, ten :: Number
zero  = [Zero]
unity = [One]
two   = [Two]
ten   = [One, Zero]
```

Here is the first basic operation, the *successor*, which we already know from Peano's axioms. To avoid confusion with the *succ* function in Haskell's *Prelude*, we will call it *next*:

```
next :: Number → Number
next [] = []
```

This is already the first important decision. We define the *next* of the empty list is the empty list. That implies that *nothing* is something different from *Zero*. We could enter difficult philosophical discussions about this statement. The decision, however, is mainly pragmatic: we need a base case for processing lists and this base case is just the empty list.

The next successors are straight forward:

```
next [Zero]  = [One]
next [One]   = [Two]
next [Two]   = [Three]
next [Three] = [Four]
next [Four]  = [Five]
next [Five]  = [Six]
next [Six]   = [Seven]
next [Seven] = [Eight]
next [Eight] = [Nine]
```

Now it gets interesting:

```
next [Nine] = [One, Zero]
```

Note that we need one more digit to represent the successor of the 10^{th} digit! The first place, read from right to left, returns to *Zero* and the second place goes up from *nothing* to *One*. This latter wording shows that our decision, concerning the empty list, is not so innocent as it may appear at the first sight!

Now we have to define how to proceed with the successor of numbers consisting of more than one digit:

$$\mathit{next}\ (\mathit{Zero} : \mathit{ds}) = \mathit{next}\ \mathit{ds}$$

The first thing we do is to check if the head is $Zero$. In this case, we just reduce to the rest of the list, that is: a leading $Zero$ does not change the value of a number. In all other cases:

$$\begin{array}{l}\mathit{next}\ \mathit{ds} = \textbf{case}\ \mathit{last}\ \mathit{ds}\ \textbf{of} \\ \qquad\qquad \begin{array}{lll}\mathit{Nine} & \rightarrow & \mathit{next}\ (\mathit{init}\ \mathit{ds}) +\!\!\!+ [\mathit{Zero}] \\ d & \rightarrow & \qquad \mathit{init}\ \mathit{ds}\ +\!\!\!+ \mathit{next}\ [d]\end{array}\end{array}$$

If the last digit of the number is $Nine$, we concatenate the successor of the number without the last digit (*init*) and [*Zero*]. The point is that the successor of $Nine$, as we have defined it above, is [*One*, *Zero*]. The last digit of the new number, hence, will be $Zero$ appended to the successor of the initial part. If the last number of the initial part is again $Nine$, we repeat the whole process on the number except the last digit. Example: the successor of the number [$Nine$, $Nine$] is

$\mathit{next}\ [\mathit{Nine}] +\!\!\!+ [\mathit{Zero}]$
$[\mathit{One}, \mathit{Zero}] +\!\!\!+ [\mathit{Zero}]$
$[\mathit{One}, \mathit{Zero}, \mathit{Zero}]$.

For the case that the last digit is not $Nine$, the process is much simpler: we just replace the last digit by its successor. The successor of [$Nine$, *Eight*], hence, is:

$[\mathit{Nine}] +\!\!\!+ \mathit{next}\ [\mathit{Eight}]$
$[\mathit{Nine}] +\!\!\!+ [\mathit{Nine}]$
$[\mathit{Nine}, \mathit{Nine}]$.

Note that this representation of numbers is not optimised for efficient processing. Haskell is not very good at accessing the last element of a list. There are many ideas to speed this up. An idea that suggests itself is to turn numbers around – relative to our usual reading direction – starting with the least siginificant digit, *e.g.* writing [*Zero*, *One*] instead of [*One*, *Zero*] to represent the number 10. We could also use a data type – such as the vector type – that allows for fast random access to all its elements. But this kind of optimisations would be better discussed in a Haskell tutorial.

The next basic operation is counting backwards. We start just as we started with *next*:

$$\begin{array}{l}\mathit{prev} :: \mathit{Number} \rightarrow \mathit{Number} \\ \mathit{prev}\ [\,] = [\,]\end{array}$$

But we now have an important difference:

$$\mathit{prev}\ [\mathit{Zero}] = \bot$$

We cannot count below *Zero*! Any attempt to do so will result in an error. We have to take care of this in all operations we will design in the future.

Counting backwards for the digits from *One* to *Nine*, however, is straight backward:

```
prev [One]   = [Zero]
prev [Two]   = [One]
prev [Three] = [Two]
prev [Four]  = [Three]
prev [Five]  = [Four]
prev [Six]   = [Five]
prev [Seven] = [Six]
prev [Eight] = [Seven]
prev [Nine]  = [Eight]
```

But what happens with numbers with more than one digit? First we ignore leading *Zero*s:

```
prev (Zero : ds) = prev ds
```

For all other cases, we use a strategy very similar to the one we used for *next*:

```
prev ds = case last ds of
            Zero -> case init ds of
                      [One] -> [Nine]
                      ds'   -> prev ds' ++ [Nine]
            d    -> init ds ++ prev [d]
```

If the last digit is *Zero*, the last digit of the new number will be *Nine* and the initial part of this number will be its predecessor. If the initial part is just [*One*], its predecessor would be *zero*, which we can ignore for this case. The predecessor of [*One*, *Zero*], hence, is [*Nine*] (not [*Zero*, *Nine*]). If the number is [*One*, *Zero*, *Zero*], the last digit will be *Nine*, which is then appended to the predecessor of [*One*, *Zero*], whose predecessor, as we know already, is [*Nine*]. The result hence is [*Nine*, *Nine*].

For the case that the last digit of the number is not *Zero*, we just append its predecessor to the initial part of the number and we are done. The predecessor of [*Nine*, *Nine*], hence, is just

```
[Nine] ++ prev [Nine]
[Nine] ++ [Eight]
[Nine, Eight].
```

Let us now look at how to add numbers. We start with the same logic we already encountered with Peano Numbers, *i.e.* we add by counting one number up and the other, simultaneously, down until we reach a base case:

```
add :: Number → Number → Number
add a []     = a
add [] b     = b
add a [Zero] = a
add [Zero] b = b
add a b      = next a ‘add‘ (prev b)
```

That is, any number added to [*Zero*] (or to the empty list []) is just that number. In all other cases, addition of two numbers a and b is defined recursively as counting a up and b down. When we hit the base case, *i.e.* b reaches [*Zero*], we have a result.

How many steps would we need to add two numbers this way? Well, that depends directly on the size of b. We will count a up, until b is [*Zero*]. For $b = 100$, we need 100 steps, for $b = 1\,000\,000$, we need 1 000 000 steps. Is there a way to improve on that? Yes, of course! We can just apply the same logic we have used for *next* and *prev*, that is adding single-digit numbers and handling the carry properly. Here is a solution:

```
add2 :: Number → Number → Number
add2 as bs = reverse $ go (reverse as) (reverse bs)
  where go [] ys             = ys
        go xs []             = xs
        go (x : xs) (y : ys) = case add [x] [y] of
                                 [_, r] → r : go xs (go ys [One])
                                 [r]    → r : go xs ys
                                 _      → ⊥
```

We see at once that the logic has changed significantly. First, we suddenly appear to care for efficiency: we reverse the lists before processing them! This, however, is not only for efficiency. We now process the number digit by digit starting with the least significant one, that is we look at the number not as a number, but as a list of digits – we exploit the structure of the data type.

Accordingly, we do not handle the base case [*Zero*] anymore, we are now concerned with the base case [], since this is the point, when we have consumed all elements of one of the lists. Until we reach the base case, we just add digit by digit. If the result is a number with two digits – note that we can never get to a number with more than two digits by adding two digits – we insert the less significant digit at the head of the list that will be created by continuing the process. We continue with the next step of *go*, but increase one of the numbers, the second one, by [One]. This is the *add carry* that takes care of the the most significant digit in the two-digit result. Again, by adding two digits, we will never get to a number with a digit greater than *One* in the first position. The greatest possible number, in fact, is $[\mathit{Nine}] + [\mathit{Nine}] = [\mathit{One}, \mathit{Eight}]$.

In the other case where the addition of the two digits results in a number with just one digit, we insert the result at the head of the list that is constructed by the regular continuation of *go* – here, we do not have to take care of any carry.

The final line of the function is just to avoid a warning from the Haksell compiler. It does not add any meaning to the definition of *add2*.

Let us look at an example, say, the addition of $765 + 998 = 1763$. We first reverse the lists, that is, we start with [*Seven*, *Six*, *Five*] and [*Nine*, *Nine*, *Eight*], but call *go* with [*Five*, *Six*, *Seven*] and [*Eight*, *Nine*, *Nine*]:

```
go [Five, Six, Seven] [Eight, Nine, Nine]
Three : go [Six, Seven] (go [Nine, Nine] [One])
Three : go [Six, Seven] (Zero : go [Nine] (go [] [One])
Three : go [Six, Seven] (Zero : go [Nine] [One])
Three : go [Six, Seven] (Zero : Zero : go [] (go [] [One]))
Three : go [Six, Seven] (Zero : Zero : go [] [One])
Three : go [Six, Seven] [Zero, Zero, One]
Three : Six : go [Seven] [Zero, One]
Three : Six : Seven : go [] [One]
Three : Six : Seven : [One]
```

The computation results in [*Three*, *Six*, *Seven*, *One*], which reversed is [*One*, *Seven*, *Six*, *Three*] and, hence, the correct resut.

So, how many steps do we need with this approach? We have one addition per digit in the smaller number plus one addition for the cases where the sum of two digits results in a two-digit number. For the worst case, we, hence, have $d + d = 2d$ steps, where d is the number of digits of the smaller number. When we add two numbers in the order of hundreds (or, more precisely, with the smaller number in the order of hundreds), we would have three additions (one for each digit of a number in the order of hundreds) plus, in the worst case, three add carries. Translated into steps of *next*, this would be for each single-digit addition in the worst case nine steps (any addition with [*Nine*]) and one *next* step per carry (since carry is always an addition of [*One*]). The worst case in terms of *next*, hence, is $9d + d = 10d$, for d the number of digits in the smaller number. For numbers in the order of millions, this amounts to $10 \times 7 = 70$ steps, compared to 1 000 000 steps for the naïve approach.

We could even go on and reduce the worst case of 9 *next* steps per addition to one single step, just by doing to the algorithm what they do to us in school: instead of using *next* for addition we can define addition tables for our 10 digits, *i.e.*

```
add [Zero] a = a
add [One] [One] = [Two]
add [One] [Two] = [Three]
add [One] [Three] = [Four]
...
```

But that approach is quite boring and, therefore, we will not go for it. Instead, we will look at subtraction. First, we implement the naïve approach that we need for subtraction of digits:

```
sub :: Number → Number → Number
sub a [Zero]             = a
sub a b | a ‘cmp‘ b ≡ LT = ⊥
        | otherwise      = prev a ‘sub‘ (prev b)
```

Subtracting *zero* from any number is just that number. For other cases, we first have to compare the numbers. If the first number is less than the second, the result is undefined for natural numbers. Otherwise, we just count the two number down by one and continue until we hit the base case.

We will look at the comparison function *cmp* below. We will first define the more sophisticated version of subtraction for numbers with more than one digit:

```
sub2 :: Number → Number → Number
sub2 as bs | as ‘cmp‘ bs ≡ LT = ⊥
           | otherwise        = clean (reverse (go (reverse as)
                                                   (reverse bs)))
  where go xs []         = xs
        go [] _          = ⊥
        go (x : xs) (y : ys)
          | y > x        = let [r] = sub [One, x] [y]
                                          in r : go xs (inc ys)
          | otherwise = let [r] = sub [x] [y] in r : go xs ys
```

As with *add*2, we reverse the lists to compute the result digit by digit starting with the least significant one using the function *go*. Note that we finally *clean* the list. *clean* removes leading *zero*s, a functionality that would certainly be very useful for real-world organisations too. Before we start the hard work entering *go*, we compare the values of the arguments and If the first one is smaller than the second one, the result is undefined.

In the *go* function, we distinguish two cases: if the first digit of the second argument is greater than that of the first one, we subtract y from $10 + x$ and increase *ys* by one. Otherwise, we just compute $x - y$.

The function *inc* is a variant of *next* for reversed numbers:

```
inc []           = [One]
inc (Nine : xs)  = Zero : inc xs
inc (x : xs)     = next [x] ++ xs
```

Applied on the empty list, *inc* yields *unity*, which is a quite different behaviour than that of *next*. Applied on a list that starts with *Nine*, we insert *Zero* as the head of the *inc*’d tail of the list. Otherwise, we just substitute the head by its *next*.

Somewhat strange might be that we need that *cmp* function – we, apparently, do not need it in other cases. The point is that we have declared that we want to derive the

Digit data type from *Ord*. With this declaration, Haskell automatically imposes the order $Zero < One < Two < \cdots < Nine$. But that would not work for lists of digits. Haskell would assume that a list like $[Nine]$ is greater than $[One, Zero]$, which, as we know, is not the case. We have to tell the compiler explicity, how we want lists of digits to be handled. This is what the *cmp* function does:

```
cmp :: Number → Number → Ordering
cmp x y = case lencmp x y of
                 GT → GT
                 LT  → LT
                 EQ → go x y
   where go [] [] = EQ
            go (a : as) (b : bs) | a > b        = GT
                                  | a < b        = LT
                                  | otherwise = go as bs
            go _ _   = ⊥
```

The function goes through all possible cases, explaining that a longer number is always the greater one and that, in the case they are equally long, one must compare all digits until one is found that is greater or smaller than the digit at the same position in the other list.

Note that we use a special function, *lencmp*, to compare the length of two lists. We do this out of purity on one hand and for efficiency on the other. It would not appear *fair* to use the Prelude function *length*, since it is expressed in terms of a number type that is already much more complete than our humble *Number*s. We could, of course, define our own *length* function, for instance:

```
len :: [a] → Number
len = foldl′ (λn _ → next n) zero
```

But, in fact, we are not too much interested in the concrete length of the two lists, we just want to know, which one, if any, is the longer one. It is not necessary to go through both lists separately in order to learn this, we can just run through both lists at the same time:

```
lencmp :: [a] → [a] → Ordering
lencmp [] []              = EQ
lencmp [] _               = LT
lencmp _ []               = GT
lencmp (_ : xs) (_ : ys) = lencmp xs ys
```

The *lencmp* function, bears a fundamental idea of comparing two sets: by assigning each member of one set to a member of the other until one of the sets is exhausted. The one that is not yet exhausted must be the greater one. Counting could be described in terms of this logic as a comparison of a set with the set of natural numbers. We assign the numbers $1, 2, \ldots$ until the first set is exhausted. The last number assigned is the size of the first set. We will learn much more about this apparently simple principle in the future.

As we are already talking about little helpers, it is the right time to introduce some fundamental list functions that we will need to elaborate on the number type later. We will need variants of *take* and *drop*:

```
nTake :: Number → [a] → [a]
nTake [Zero] _   = []
nTake _ []       = []
nTake n (x : xs) = x : nTake (prev n) xs
```

```
nDrop :: Number → [a] → [a]
nDrop [Zero] xs  = xs
nDrop _ []       = []
nDrop n (_ : xs) = nDrop (prev n) xs
```

Very useful will be a function that turns all elements of a list into *Zero*s:

```
toZero :: [a] → [Digit]
toZero = map (const Zero)
```

We should also introduce the enumeration function that facilitates list definition, *i.e.* that gives us a list for a range of numbers of the form $[1 \mathinner{..} 9]$ or, in terms of the *Number* type, $[[One] \mathinner{..} [Nine]]$:

```
enum :: Number → Number → [Number]
enum l u | l `cmp` u ≡ GT = []
         | otherwise      = go l u
  where go a b | a `cmp` b ≡ EQ = [a]
               | otherwise      = a : go (next a) b
```

Finally, we also need the function *clean*, which is defined as:

```
clean :: Number → Number
clean [Zero]      = [Zero]
clean (Zero : ds) = clean ds
clean ds          = ds
```

We, hence, leave the number [*Zero*] untouched. If the number starts with the digit *Zero*, but has more than just that one number, we ignore this leading *Zero* and continue with the remainder of the list. (Note that, since the case of a list that consists of only the digit *Zero* is already handled in the first case, *ds* in the second case will never be the empty list!) Finally, a list that does not start with *Zero* is just given back as it is.

Now, let us turn to the model for our number type, that is how we interpret a list of digits. There are many ways to interpret numbers. A somewhat natural way is to indicate a function that, for any list of *Digit*s, gives us the numerical value of the number we intend to represent with this list. The, perhaps, most obvious way to do so is to convert the list of *Digit*s into a string and then to read this string in again as integer. We would define a conversion function of the form

```
toString :: Number → String
toString = map toChar
  where toChar Zero = '0'
        toChar One  = '1'
```

and so on. But this approach is not very interesting. It does not give us any insight. What we would like to have instead is a model that explains how numeral systems work in general. The key to understand how such a model can be devised is to see that our system consists of 10 symbols. With one of these symbols, we, hence, can represent 10 different numbers. With two of these symbols, we represent 10×10 numbers, that is the numbers $0 \dots 9$ plus the numbers $10 \dots 19$ plus the numbers $20 \dots 29$ plus ... the numbers $90 \dots 99$. With three of these symbols, we then can represent $10 \times 10 \times 10$ numbers, namely the numbers $0 \dots 999$ and so on. In other words, the *weight* of a digit in a number represented in a numeral system with b symbols corresponds to a power of b, which we therefore call the *base* of that numeral system. A numeral system with 2 symbols would have the base 2, the weight of each digit would therefore be a power of 2. A numeral system with 16 symbols has the base 16 and the weight of each digit would be a power of 16.

The exponent, that is to which number we raise the base, is exactly the position of the digit in a number if we start to count positions with 0. The number 1763 has the value: $1 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 3 \times 10^0 = 1000 + 700 + 60 + 3$:

3	2	1	0
1	7	6	3

We could devise a data type that represents those *weighted* digits nicely as:

```
type WDigit  = (Number, Digit)
type WNumber = [WDigit]
```

The *WDigit* is a tuple of *Number* and *Digit*, where the number is the exponent to which we have to raise the base. We can convert a *Number* easily to a *WNumber* by:

```
weigh :: Number → WNumber
weigh = go [Zero] ∘ reverse
  where go _ []       = []
        go n (d : ds) = (n, d) : go (next n) ds
```

The *weigh* function reverses the input in order to start with the least significant digit and then just passes through this list adding the exponent incrementing it by one in each step. Note that the order of a *WNumber* does not matter anymore, because the decisive information that is encoded in the position of each digit is now made explicit in the exponent.

The inverse of this function is:

```
unweigh :: WNumber → Number
unweigh = reverse ∘ map snd ∘ complete [Zero] ∘ sortW
  where complete _ [ ] = [ ]
        complete n ((e, d) : xs)
          | n ‘cmp‘ e ≡ LT = (n, Zero) : complete (next n) ((e, d) : xs)
          | otherwise      = (e, d)    : complete (next n) xs
sortW :: WNumber → WNumber
sortW = sortBy (λx y → fst x ‘cmp‘ fst y)
```

We, first, sort the components of the *WNumber* in ascending order according to their exponents. We, then, *complete* the *WNumber*, *i.e.* we fill in *Zero*s for missing exponents such that the resuling *WNumber* has a component for every exponent from 0 to the greatest one present. From this list, we extract the digits and reverse the result.

To build the model, we still need a function that converts digits into one-digit integers. This is straight forward:

```
digit2Int :: Digit → Int
digit2Int Zero  = 0
digit2Int One   = 1
digit2Int Two   = 2
digit2Int Three = 3
digit2Int Four  = 4
digit2Int Five  = 5
digit2Int Six   = 6
digit2Int Seven = 7
digit2Int Eight = 8
digit2Int Nine  = 9
```

To convert a *Number* to an *Integer*, we first convert the *Number* to a *WNumber* and then convert the *WNumber* to an *Integer*:

```
n2Integer :: Number → Integer
n2Integer [ ]  = 0
n2Integer [n] = fromIntegral (digit2Int n)
n2Integer ns  = w2Integer    (weigh ns)
```

As a convention, we convert the empty list into 0. (We could raise an error for this case, but that does not appear to be necessary or even useful.) A one-digit number is simply converted by converting its single digit. Since *digit2Int* converts a digit to an *Int*, but we now want an *Integer*, we still have to call *fromIntegral* on the result, to convert from *Int* to *Integer*.

Numbers with many digits are converted to *WNumber* using *weigh* and then converted to *Integer* using *w2Integer*:

```
w2Integer :: WNumber → Integer
w2Integer          = sum ∘ map conv
  where conv w = let x = n2Integer (fst w)
                     d = fromIntegral (digit2Int (snd w))
                 in  d ∗ 10 ↑ x
```

Weighted numbers are converted to *Integer* by summing up the single values of the digits, which are calculated in terms of powers of 10: $d \times 10^x$, where d is the digit converted to *Int* by *digit2Int* and then converted to *Integer* by *fromIntegral*. x is the exponent converted to *Integer* using *n2Integer*.

This looks like an infinite regress where we convert the exponent of the weighted number, which is a *Number*, to an Integer using *n2Integer*, which then calls *w2Integer*, which again calls *n2Integer* to convert the exponent, which, again, calls *w2Integer* and so on.

It is indeed very well possible that we have extremely large numbers with exponents that are many digits long, but even the greatest number will finally converge to an exponent that is smaller then 10. The incredibly large number $10^{100000000000}$, for example, has an exponent with 12 digits, which, represented as a *Number*, is

[*One*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*, *Zero*].

The greatest exponent in this number, the exponent of the leading *One*, however, has just two digits: [*One*, *One*], which, in the next conversion step, reduces to [*One*] for the most significant digit and, hence, will be converted immediately to 1.

Of course, we do not need the data type *WNumber* for this conversion. We could very well have converted a number by reverting it and then pass through it with an inreasing *Integer* exponent starting from 0. The detour through weighted numbers, however, is a nice illustration of the model for our number system, and, perhaps, there will be use for this or a similar data type later during our journey.

3.4. Multiplication

As addition can be seen as a repeated application of counting up, multiplication can be seen as repeated addtion. A naïve implementation of multiplication, hence, is:

```
mul :: Number → Number → Number
mul _ [Zero] = [Zero]
mul [Zero] _ = [Zero]
mul a [One]  = a
mul [One] b  = b
mul a b      = a `add2` (a `mul` (prev b))
```

Notable, here, is that we have more base cases than with addition: Any number multiplied by *zero* is just *zero* and any number multiplied by *unity* is just that number. From here on, we add a to a and count b down simultaneously, until b reaches the base case $[One]$.

This simple implementation is not optimal in terms of computation complexity: we need b steps to multiply a and b. For a multiplication of two numbers in the range of millions, we need millions of single additions. As with addition, we can improve on this by multiplying digit by digit or, more precisely, by multiplying all digits of the first number by all digits of the second number. This, however, is somewhat more complicated than in the case of addition, because multiplication has effect on the weight of the digits. On two one-digit numbers, weight has no impact, since the weight of each digit is just 0: $(2 \times 10^0) \times (3 \times 10^0) = 6 \times 10^{0+0} = 6 \times 1 = 6$. But, if 2 and 3 above were digits of a number with more than one digit and, themselves not the least significant digits, *e.g.* 20×30, then we would have something of the form: $(2 \times 10^1) \times (3 \times 10^1) = 6 \times 10^2 = 600$.

We, therefore, have to take the weight of the digits into account. But what is the best way to do so? We could of course use the weighted number type we already defined in the previous section. The disadvantage of this approach, however, is that we have to perform additional arithmetic on the weight, potentially searching for equal weights in the resulting number or reordering it to bring equal weights together. We can avoid this overhead by reflecting the weight in numbers, simply by appending $n - 1 + m - 1$ *Zero*s to the result of multiplying the n^{th} digit of one number with the m^{th} digit of the other one and, eventually, adding up all these components.

We first implement a function that multiplies a digit with all digits of a number appending *Zero*s to each result and adding them up:

```
mul1 :: Digit → Number → [Digit] → Number
mul1 _ []          _  = zero
mul1 x (y: _)      [] = [x] `mul` [y]
mul1 x (Zero : ys) zs = mul1 x ys (tail zs)
mul1 x (y : ys)    zs = add2 ([x] `mul` [y] ++ zs)
                             (mul1 x ys (tail zs))
```

If the number is exhausted, *i.e.* if we have already multiplied all digits, the result is just *zero*. If the *Zero*s have been exhausted, whatever remains from the number, we multiply x with the head of that rest. We ignore *Zero*s in the number, but make sure to consider their weight by reducing the trail of *Zero*s by one in the continuation. In all other cases, we multiply x and the first digit in the number using the simple *mul* and appending the *Zero*s to the result. This result is then added to the result of the recursion of *mul1* with the tail of the number and the tail of the *Zero*s.

This function is now *mapped* on *Number*:

```
mulN :: Number → Number → [Digit] → Number
mulN []          _ _  = zero
mulN (x: _)      b [] = mul1 x b []
mulN (Zero : xs) b zs = mulN xs b (tail zs)
mulN (x : xs)    b zs = add2 (mul1 x b zs)
                             (mulN xs b (tail zs))
```

If the first number is exhausted, we just return *zero*. If the *Zero*s are exhausted, we apply *mul1*, *i.e.* we multiply one digit with *b*, and terminate. Note that the *Zero*s should be exhausted only if there is just one digit left in the numbers. Again, we ignore *Zero*, but respect its weight. In all other cases, we apply *mul1* on the first digit of the first number and add the result with the recursion on the tail of the first number and the tail of *Zero*s.

Finally, we apply *mulN* on two numbers creating the trail of *Zero*s:

```
mul2 :: Number → Number → Number
mul2 [] _ = zero
mul2 _ [] = zero
mul2 a b  = mulN a b ((toZero $ tail a) ++
                      (toZero $ tail b))
```

We handle the cases where one of the numbers is the empty list explicitly to avoid problems with the call of *tail* later on. We then call *mulN* for *a* and *b* and the trail of *Zero*s that results from converting all digits but one in *a* and *b* to *Zero*.

Note that this is exactly what we do, when we elaborate a multiplication with pen and paper. If we multiplied, say, 13×14, we would write the partial results aligned according to the number of zeros they would have:

13×14
$1 \times 1 = 100$
$1 \times 4 = 040$
$3 \times 1 = 030$
$3 \times 4 = 012$

Now we add up the partial results:

$100 + 040 = 140$
$030 + 012 = 042$
$140 + 042 = 182$

The grouping of additions chosen here corresponds to the additions performed in *mul1* and *mulN*: The first two additions are performed in *mul1*, the last line is done in *mulN*.

Let us look at how *mulN* works for [*One*, *Three*] × [*One*, *Four*]. We start with

```
mulN (One : [Three]) [One, Four] [Zero, Zero] =
add2 (mul1 One [One, Four] [Zero, Zero]) (mulN [Three] [One, Four] [Zero]).
```

The first term of *add2* is

$$\begin{array}{l}\mathit{mul1}\ \mathit{One}\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}, \mathit{Zero}] = \\ \mathit{add2}\ (([\mathit{One}]\ \text{`}\mathit{mul}\text{`}\ [\mathit{One}]) \mathbin{+\!\!+} [\mathit{Zero}, \mathit{Zero}])\ (\mathit{mul1}\ \mathit{One}\ [\mathit{Four}]\ [\mathit{Zero}]).\end{array}$$

The first term, here, reduces to

$$[\mathit{One}] \mathbin{+\!\!+} [\mathit{Zero}, \mathit{Zero}] = [\mathit{One}, \mathit{Zero}, \mathit{Zero}],$$

which corresponds to **100** in the paper example above. The second term reduces to

$$\begin{array}{l}\mathit{mul1}\ \mathit{One}\ (\mathit{Four} : [\,])\ [\mathit{Zero}] = \\ \mathit{add2}\ (([\mathit{One}]\ \mathit{mul}\ [\mathit{Four}]) \mathbin{+\!\!+} [\mathit{Zero}])\ (\mathit{mul1}\ [\mathit{One}]\ [\,]\ [\,]),\end{array}$$

which, in its turn, is just

$$\mathit{add2}\ [\mathit{Four}, \mathit{Zero}]\ [\mathit{Zero}] = [\mathit{Four}, \mathit{Zero}]$$

and corresponds to **40** in the manual calculation. We, hence, have

$$\mathit{add2}\ [\mathit{One}, \mathit{Zero}, \mathit{Zero}]\ [\mathit{Four}, \mathit{Zero}] = [\mathit{One}, \mathit{Four}, \mathit{Zero}]$$

at the end of the first round of *mul1*. This is the same result as we obtained in the first addition step in the manual process, *i.e.* **140**. Returning to the first equation, we now have:

$$\begin{array}{l}\mathit{mulN}\ (\mathit{One} : [\mathit{Three}])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}, \mathit{Zero}] = \\ \mathit{add2}\ ([\mathit{One}, \mathit{Four}, \mathit{Zero}])\ (\mathit{mulN}\ [\mathit{Three}]\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}]).\end{array}$$

The second term of *add2*, here, produces:

$$\begin{array}{l}\mathit{mulN}\ (\mathit{Three} : [\,])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}] = \\ \mathit{add2}\ (\mathit{mul1}\ \mathit{Three}\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}])\ (\mathit{mulN}\ [\,]\ [\mathit{One}, \mathit{Four}]\ [\,]).\end{array}$$

The first term, the call to *mul1*, is:

$$\begin{array}{l}\mathit{mul1}\ \mathit{Three}\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}] = \\ \mathit{add2}\ (([\mathit{Three}]\ \text{`}\mathit{mul}\text{`}\ [\mathit{One}]) \mathbin{+\!\!+} [\mathit{Zero}])\ (\mathit{mul1}\ \mathit{Three}\ (\mathit{Four} : [\,])\ [\,]).\end{array}$$

The first term of this addition is $[\mathit{Three}, \mathit{Zero}]$, which corresponds to the same result **30** we had above in the third step of the manual multiplication, and the second term is:

$$\mathit{mul1}\ [\mathit{Three}]\ (\mathit{Four} : [\,])\ [\,] = [\mathit{Three}]\ \text{`}\mathit{mul}\text{`}\ [\mathit{Four}] = [\mathit{One}, \mathit{Two}].$$

The result **12** we obtained before. Going back, we now have:

$$\begin{array}{l}\mathit{mul1}\ [\mathit{Three}]\ (\mathit{One} : [\mathit{Four}])\ [\mathit{Zero}] = \\ \mathit{add2}\ [\mathit{Three}, \mathit{Zero}]\ [\mathit{One}, \mathit{Two}] = [\mathit{Four}, \mathit{Two}]\end{array}$$

We now have the result of the second addition step in the paper multiplication, *i.e.* **42**. and, returning to the first equation, we get the final result:

$$\begin{array}{l}\mathit{mulN}\ (\mathit{One} : [\mathit{Three}])\ [\mathit{One}, \mathit{Four}]\ [\mathit{Zero}, \mathit{Zero}] = \\ \mathit{add2}\ [\mathit{One}, \mathit{Four}, \mathit{Zero}]\ [\mathit{Four}, \mathit{Two}] = [\mathit{One}, \mathit{Eight}, \mathit{Two}].\end{array}$$

This is the last line of the addition: $140 + 42 = \mathbf{182}$.

How many steps do we need for multiplication with this approach? We, first, multiply all digits of one number with all digits of the other number and, thus, perform $n \times m$ one-digit multiplications, where n and m are the numbers of digits of the first and the second argument respectively. We then add all the $n \times m$ numbers together, resulting in $n \times m - 1$ multi-digit additions. Most of the multi-digit additions, though, add *Zero*s, which is just one comparison and, hence, quite unexpensive. We have, however, many of those simple steps, because we add numbers of the size $n + m - 1, n + m - 2, \dots, 1$. This is the addition of all numbers from 1 to $n + m$, a type of problems, we will study in the next chapter.

Anyhow, the cost for *mul2* grows only in the size of the arguments, whereas the naïve *mul* grows directly in the value of the second number. The number of steps is $nmp+(nm-1)a$, where p is the cost for a single-digit multiplication and a that of an addition. For very, very large numbers, say, numbers with thousands or millions of digits, the approach, still, is too slow. There are many ways to multiply more efficiently, but that is not our focus here.

Multiplication appears to be such a tiny simple device, but it introduces huge complexity. If we just look at the patterns that *mul2* produces when processing two numbers $[a, b]$ and $[c, d]$: $[ac + ad + bc + bd]$, we see that multiplication is intimately involved with problems of combinatorics, which too will be a major topic of the next chapter. Imagine the multiplication of a number with itself, *i.e.* where c and d equal a and b, respectively:

$$[a, b] \times [a, b] = [aa + ab + ba + bb] = [a^2 + 2ab + b^2]. \tag{3.1}$$

Indeed, multiplying $[One, Two]$ with itself results in $[One, Four, Four]$ and $[One, Three]$ in $[One, Six, Nine]$. This pattern plays a role in many branches of mathematics, like algebra, combinatorics and probability theory, and is truly one of the most important facts you can learn about mathematics. Should equation 3.1 not be familiar to you already, you definitely should memorise it. The tiny device of multiplication, one could contemplate, is a focal point of many complications we will encounter on our journey – and this appears to me as one of the characteristics of mathematics: that small problems, such as multiplication, thought through, develop unforeseen impact on apparently completely different subjects.

A nice illustration of the patterns created by multiplication is the results of squaring numbers that consist only of 1s. Have a look at the following pyramid:

$$1 \times 1 = 1$$
$$11 \times 11 = 121$$
$$111 \times 111 = 12321$$
$$1111 \times 1111 = 1234321$$
$$11111 \times 11111 = 123454321$$

It is as if the digit in the centre of the number on the right-hand side of the equations wanted to tell us the size of the factors used to create it. When we try to fool the numbers, leaving some 1s out in one of the factors, they realise it immediately, and come up with "damaged" results like

$$1 \times 11 = 11$$
$$11 \times 111 = 1221$$
$$11 \times 1111 = 12221$$
$$11111 \times 1111111111 = 12345555554321.$$

Now, the central digits in the result tell us the size of the smaller number and their repetition tells us the difference to the greater factor, which is exactly one less than the number of repetitions.

3.5. Division and the Greatest Common Divisor

It is now time to introduce Euclid. Unusually little is known about this author. Important scholars of the time (about 300 BC) are usually mentioned by name in philosophical texts of other authors and often with some biographical detail. In the case of Euclid, this is different. Euclid is rarely mentioned by name – and when it happens, he is confused with an earlier philosopher of the same name – and nothing is told about his life but the fact that he was active in Alexandria for some time. This is particularly strange, since Euclid's work had a tremendous influence on the antiquity and on through the middle ages up to our days. This has led to the conjecture that Euclid was not a person, but a group of scholars at the university or library of Alexandria. This idea may be inspired by similar conjectures concerning the "person" of Homer or by the existence of groups named after fictional characters in later times like, in the 20^{th} century, the "Association of collaborators of Nicolas Bourbaki", a highly influential group of mathematicians dedicated to the formalisation of mathematics. Nicolas Bourbaki, even though he had an office at the École Normale Supérieure for some time, did not exist. He is a fictional character whose name was used for the publications of the Bourbaki collective.

Euclid – who or whatever he was – is the author of the *Elements*, the mother of all axiomatic systems and, certainly, one of the greatest intellectual achievements of the antiquity. The *Elements* lay out the acient knowledge on geometry, arithmetic and number theory in fifteen books following a rigid plan starting with axioms, called “postulates”, followed by theorems and their proofs based only on the axioms. There are some inaccuracies in the choice of the axioms and not all proofs are rock-solid according to modern standards. But, anyway, the rigidity of the Elements was not achieved again before the 19^{th} century, perhaps with the *Disquisitiones Arithmeticae* by 21-year-old Carl Friedrich Gauss.

Here, we are interested mainly in some of the content of book 7, which deals with issues of arithmetic and elementary number theory, in particular division and the greatest common divisor. According to Euclid, division solves equations of the form

$$a\ div\ b = q + r, \tag{3.2}$$

and fulfils the constraint

$$a = qb + r, 0 \leq r < b. \tag{3.3}$$

There is a kind of mismatch between this notion of division, usually called *division with remainder*, and multiplication in that multiplication of any two natural numbers results in a natural number, whereas division with remainder results in two numbers, the *quotient* q and the *remainder* r. The division of two numbers that are *divisible*, *i.e.* the division leaves no remainder, is just a special case of this operation like in $9\ div\ 3 = 3+0$. In other cases, this does not work: $8\ div\ 3 = 2 + 2$, since $2 \times 3 + 2 = 8$. We already have seen such a mismatch with addition and subtraction: the addition of any two natural numbers always produces a natural number; subtraction, however, does only produce a natural number when its second term is less than or, at most, equal to the first term. This will be an important topic in the progress of our investigations.

Euclid’s algorithm to solve the equation goes as follows: Division by zero is not defined. Division of zero by another number (not zero) is zero. Otherwise, starting with the quotient $q = 0$ and the remainder $r = a$, if the remainder r is less than the divisor b, then the result is (q, r). Otherwise, we decrement the remainder by b and increment q by one:

```
quotRem :: Number → Number → (Number, Number)
quotRem _ [Zero] = error "division by zero"
quotRem [Zero] _ = (zero, zero)
quotRem a [One] = (a, zero)
quotRem a b        = go a zero
  where go r q | r ‘cmp‘ b ≡ LT = (q, r)
               | otherwise        = go (r ‘sub2‘ b) (next q)
```

As you should realise at once, this algorithm is not efficient for large numbers a. If a is much larger than b, we will have to subtract lots of bs from it. In fact, the complexity of this algorithm is $\lfloor a/b \rfloor$, since we need $\lfloor a/b \rfloor$ steps to bring a down to an r that is smaller than b. The complexity of the algorithm, hence, equals (a part of) its result!

As usual, we can improve by taking the structure of the numbers into account, namely by operating on digits instead of whole numbers. Have a look at the following, admittedly, scary-looking listing:

```
quotRem2 :: Number → Number → (Number, Number)
quotRem2 _ [Zero] = error "division by zero"
quotRem2 [Zero] _ = (zero, zero)
quotRem2 a [One]  = (a, zero)
quotRem2 a b      = go zero [ ] a
  where go q [ ] [ ] = (clean q, zero)
        go q c [ ]   = (clean q, clean c)
        go q c r     = let x = clean (c ++ [head r])
                           y = tail r
                       in if x `cmp` b ≡ LT
                          then go (q ++ zero) x y
                          else let (q', r') = quotRem x b
                                   r2 | r' ≡ zero = [ ]
                                      | otherwise = r'
                               in go (q ++ q') r2 y
```

We start, as usual, with the base cases: division by zero and not defined; zero divided by something else is zero. A number divided by one is just that number.

For all other cases, we call *go* with *zero* as quotient and a as remainder. There is an additional parameter, c, which takes care of carries. If we have exhausted, both the carries and the remainder, then the result is just $(q, zero)$, *i.e.* we have no remainder. If the remainder is exhausted, but not the carries, the carries together are the remainder. Otherwise, we proceed as follows: We take the head of of the remainder and concatenate it to previous carries starting with the empty list. If this number is less than b, we append a *Zero* to q and continue with x as carry and the *tail* of r. Note that, if this happens on the first digit, the *Zero*s appended to q will be cleaned off later. Only *Zero*s between digits are taken into account. This is exactly what we do, when we divide with pencil and paper: when, during the process, the next number in a cannot be divided by b, we append a zero to the partial result obtained so far and append the next number of a to the remainder of the previous calculation.

Otherwise, if x is not less than b, we divide these two numbers using the naïve *quotRem*. The quotient resulting from the application of *quotRem* is appended to the previous result q. The remainder, if not zero, is carried over. Since *quotRem* is applied, as soon as we arrive at a number that is equal to or greater than b appending one digit of a after the other, this number is at most 9 times as big as b. In other words, *quotRem* in this context, will never need more than 9 steps. Nevertheless, *quotRem* is the bottleneck of this implementation. With lookup tables for one-digit divisions, we could reach a significant speed-up. But optimising, again, is not our prime concern here. Therefore, we will stick with this suboptimal solution.

An important aspect of the algorithm is that we chop off leading *Zero*s, whenever we go to use a sequence of digits as a number, in particular before we return the result and before calling *quotRem*. The algorithm handles numbers as sequence of digits that are as such meaningless. But whenever it operates on those sequences it takes care of handling them as proper numbers.

Let us look at a simple example, say, $[One, Two, Three]$ divided by $[Six]$. We start with

$go\ [Zero]\ []\ [One, Two, Three]$

and compute x as $clean\ ([] +\!\!+ [One])$ and y as $[Two, Three]$. Since x, which is $[One]$, is less than b, $[Six]$, we continue with

$go\ ([Zero] +\!\!+ [Zero])\ [One]\ [Two, Three]$.

This time x is $clean\ ([One] +\!\!+ [Two])$ and y is $[Three]$. x now is greater than b and therefore we compute

$(q', r') = quotRem\ [One, Two]\ [Six]$

where q' is $[Two]$ and r' is $[Zero]$. We then continue with

$go\ ([Zero, Zero] +\!\!+ [Two])\ []\ [Three]$

and compute x as $[Three]$ and y as $[]$. Since x, again, is less than b, we continue with

$go\ ([Zero, Zero, Two] +\!\!+ [Zero])\ [Three]\ []$,

which is the second base case of *go* leading to

$(clean\ [Zero, Zero, Two, Zero], clean\ [Three])$,

which in its turn is just $([Two, Zero], [Three])$ expressing the equation $6 \times 20 + 3 = 123$.

There are many interesting things to say about division and especially about the concept of the remainder. First, the remainder is an indicator for *divisibility*. A number b is said to divide a number a or a is divisible by b, $b \mid a$, if $a\ div\ b = (q, 0)$, *i.e.* if the remainder of the Euclidian division is 0. In Haskell, we can define the remainder as:

```
rem :: Number → Number → Number
rem a b = snd (quotRem2 a b)
```

The quotient, correspondingly, is

```
div :: Number → Number → Number
div a b = fst (quotRem2 a b)
```

Divisibility, then, is:

```
divides :: Number → Number → Bool
divides a b | rem b a ≡ zero = True
            | otherwise      = False
```

There are some rules (valid for natural numbers) that can be defined on divisibility, namely: For all numbers a: $1 \mid a$, that is: 1 divides all numbers, since $a\ \mathit{div}\ 1 = (a, 0)$.

It holds also that $a \mid b \wedge b \mid c \rightarrow a \mid c$. In other words: if a divides b and b divides c, then a also divides c. (The symbol "$\wedge$" means "AND" here.) This is because, if b divides c, then c is a multiple of b and, if a divides b, then b is a multiple of a and, in consequence, c is also a multiple of a. Any number divisible by 4, for instance, is also divisible by 2, since $2 \mid 4$.

Furthermore, if $a \mid b$ and $b \mid a$, then we can say that $a = b$, since, if a were greater than b, then a would not divide b and vice versa.

An interesting – and important – equality is also $a \mid b \wedge a \mid c \rightarrow a \mid (b + c)$. This rule says that the sum of any two numbers b and c, both divisible by another number a is also divisible by a. For the special case $a = 2$, this rule says that the sum of two even numbers is also even: $4 + 6 = 10$, $50 + 28 = 78$, $1024 + 512 = 1536$, … This is true in general for all numbers a, *e.g.* 5: $10 + 15 = 25$, which is $2 \times 5 + 3 \times 5 = 5 \times 5$, or $35 + 625 = 660$, which is $7 \times 5 + 125 \times 5 = 132 \times 5$. We can go even further and say $a \times b + a \times c = a \times (b + c)$. This is called the distributive law and we have already used it implicitly when defining multiplication. We will come back to it very soon.

The remainder gives rise to an especially interesting concept, the concept of arithmetic *modulo n*. The term modulo refers just to the remainder of the Euclidian division. Most implementations in programming languages, including Haskell, distinguish the operator *mod* and *rem* according to the *signedness* of dividend and divisor. For the moment, that is not relevant for us, since we are working with natural numbers only, so, for the moment, we will treat *mod* and *rem* as being the same concept.

The most common example of modulo arithmetic is time measured with a 12 or 24 hours clock. At midnight, one can say it is 12 o'clock; since $12 \bmod 12 = 0$, we can also say, it is 0 o'clock. With the 24 hours clock, one hour after noon is 13:00 o'clock. $13 \bmod 12 = 1$, 13, thus, is just 1 in the 12 hours clock. This principle works for arbitrary large numbers, *e.g.* 36 is 12, since $36 \bmod 12 = 0$ and, since $36 \bmod 24 = 12$, we can say it is noon. 500 is 8 in the evening, since $500 \bmod 24 = 20$ and $20 \bmod 12 = 8$. With modular arithmetic, arbitrary large numbers modulo n are always numbers from 0 to $n - 1$ and any operation performed on numbers modulo n results in a number between 0 and $n - 1$. This apparently trivial fact is of huge importance. We will come back to it over and over again.

Especially interesting for programmers is arithmetic modulo 2, because any operation has either 0 or 1 as result, *i.e.* the vocabulary of binary number representation. Indeed, addition of the numbers 0 and 1 modulo 2 is just the *exclusive or* (XOR) operation: $0 + 0 = 0 \mod 2$, $1 + 0 = 1 \mod 2$, $1 + 1 = 0 \mod 2$, since $1 + 1 = 2$ and $2 \bmod 2 = 0$.

The XOR operation gives the same results: $0 \oplus 0 = 0$, $1 \oplus 0 = 1$, $1 \oplus 1 = 0$. Multiplication modulo 2 is equivalent to AND: $0 \times 0 = 0 \mod 2$, $0 \times 1 = 0 \mod 2$, $1 \times 1 = 1 \mod 2$. The truth values of the formula $p \wedge q$ are shown in the table below:

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

One of the fundamental tools developed in the Elements is *gcd*, the *greatest common divisor*. As the name suggests, the *gcd* of two numbers a and b is the greatest number that divides both, a and b.

The algorithm given in the Elements is called *Euclidian algorithm* and is used with a small, but important variation until today. The original algorithm goes as follows: the gcd of any number a and 0 is a; the gcd of any number a with any number b is $\gcd(b, a - b)$, where $0 < b \leq a$. If $b > a$, we just turn the arguments around: $\gcd(b, a)$.

For large numbers, this is not efficient, especially, if a is much greater than b. The remarks on modulo above, however, hint strongly at a possible optimisation: the use of the remainder operation instead of difference:

```
gcd :: Number → Number → Number
gcd a [Zero] = a
gcd a b      = gcd b (a ‘rem‘ b)
```

Let us look at some examples:

gcd [*Nine*] [*Six*] = *gcd* [*Six*] ([*Nine*] ‘*rem*‘ [*Six*]),

which is *gcd* [*Six*] [*Three*], which, in its turn, is

gcd [*Three*] ([*Six*] ‘*rem*‘ [*Three*] = [*Zero*])

and, hence

gcd [*Three*] [*Zero*] = [*Three*].

More complicated is the *gcd* of [One,One] and [Six]:

gcd [*One*, *One*] [*Six*] = *gcd* [*Six*] ([*One*, *One*] ‘*rem*‘ [*Six*]),

which is

gcd [*Six*] [*Five*] = *gcd* [*Five*] ([*Six*] ‘*rem*‘ [*Five*]),

which is

gcd [*Five*] [*One*] = *gcd* [*One*] ([*Five*] ‘*rem*‘ [*One*]), which leads to

gcd [*One*] [*Zero*] = [*One*].

It is noteworthy that the algorithm always terminates. This is true because, since *rem* always reduces b to a value between *zero* and $a - 1$ and, with a getting smaller and smaller, we must at some point reach either *unity* (when b does not divide a) or *zero* (when b does divide a). If we reach *zero*, we have a result; otherwise, we will reach *zero* in the next step, because *unity*, as we have already discussed, divides any number.

Furthermore, if a is the smaller number, *gcd* will just flip the arguments, *e.g.* *gcd* 10 100 = *gcd* 100 (10 ‘*rem*‘ 100) and, since 10 *div* 100 = $(0, 10)$, this corresponds to *gcd* 100 10.

We will analyse the running time of *gcd* later in chapter 3. For now, it may suffice that each step reduces the problem to $a \bmod b$, which is in the range of $0 \dots b - 1$, while, with the original algorithm, the problem is reduced only to $a - b$ per step. With large numbers and, in particular, with a huge difference between a and b, this reduction is quite small. With the reduction by $a \bmod b$, the difference between the numbers and even the size of a do not matter. That is an effect of modular arithmetic.

An important insight related to the *gcd*, is *Euclid's lemma*, which states that if a divides cb, then a must share common factors with c or b. This is easy to see, since, that a divides cb means that there is a number n, such that $na = cb$. This number is $n = cb/a$. If a and cb did not share common factors, then cb/a would not be a natural number. For example 10 and 7 do not share factors with 3; there is thus no natural number n, such that $3n = 7 \times 10$. With 6 instead of 7, however, there is a common factor, namely 3 itself. Therefore, we can solve $3n = 6 \times 10 = 60$, simply by dividing 3 on both sides of the equation: $n = 60/3 = 20$.

Finally, we should mention a cousin of *gcd*, the *least common multiple*, *lcm*, the smallest number that is a multiple of two numbers, a and b. The obvious multiple of two numbers is the product of these numbers $a \times b$. But there may be a smaller number c, such that $a \mid c \wedge b \mid c$. How can we find that number? Well, if a and b have a gcd that is not 1, then any number divisible by a and divisible by b is also divisible by $\gcd(a, b)$. The product of a and b, hence, is divisible by $\gcd(a, b)$ and, since the *gcd* is the common divisor that reduces the product $a \times b$ most, that quotient must be the least common multiple, *i.e.*

$$lcm(a, b) = \frac{a \times b}{\gcd(a, b)}. \tag{3.4}$$

3.6. Powers, Roots and Logarithms

Multiplication can be seen as a kind of higher-order addition: one of the factors tells us how often we want to add the second factor to itself: $a \times b = b + b \dots$ This relation can be expressed nicely with the summation notation $\sum$:

$$a \times b = \sum_{i=1}^{a} b$$

For instance, 2×3 is $\sum_{i=1}^{2} 3 = 3 + 3 = 6$ and 1×3 would just be $\sum_{i=1}^{1} 3 = 3$. For $a = 0$, summation is defined as 0.

In Haskell, for any a of type class *Num*, this is implemented as $sum :: [a] \rightarrow a$, which takes an argument of type $[a]$ and returns the sum of all elements in the input list. For our number type (which we have not yet defined as *Num*), this could be:

```
summation :: [Number] → Number
summation = foldr add2 zero
```

From this definition of multiplication as repeated addition, we can go further. We can introduce an operation that repeats multiplication of a number with itself. This operation is called *power*: $a^b = a \times a \times \ldots$ and can be captured with the product notation:

$$a^b = \prod_{i=1}^{b} a$$

a^2, for instance, is $\prod_{i=1}^{2} a = a \times a$. For $b = 0$, the product is defined as 1.

In Haskell, the product for any a of type class *Num* is implemented as $product :: [a] \rightarrow a$. For our number type, we could define:

```
nProduct :: [Number] → Number
nProduct = foldr mul2 unity
```

We can define *power* as:

```
power :: Number → Number → Number
power _ [Zero] = unity
power a [One]  = a
power a b      = a `mul2` power a (prev b)
```

This algorithm, of course, is not efficient, since it needs b steps to calculate the b^{th} power of any number. A common trick to accelerate the algorithm is *exponentiation by squaring* where we reduce b faster than by just decrementing it by one. Indeed, when we exponentiate a number with an even number b, the result is $a^{2^{\frac{b}{2}}}$. What about odd bs? In this case, we reduce b by one, then we have an even number in the exponent, and multiply a once more: $a \times a^{2^{\frac{b}{2}}}$. With this algorithm, we need, instead of b steps, a logarithmic amount of steps (log base 2), which we will discuss in a second, plus one extra multiplication, when b is odd. In Haskell, this variant of power could be implemented as follows:

```
power2 :: Number → Number → Number
power2 _ [Zero] = unity
power2 a [One]  = a
power2 [One] _  = unity
power2 a b      = case b ‘quotRem2‘ two of
                    (q, [Zero]) → power2 (a ‘mul2‘ a) q
                    (q, _)      → a ‘mul2‘
                                    power2 (a ‘mul2‘ a) q
```

From *power*, we can go on further, introducing an operator that operates on powers, and, indeed, there is Knuth's *up-arrow* notation: $a \uparrow\uparrow b = a^{b^{\cdots}}$, which indicates that we raise a b times to the b^{th} power. When we have defined this, we can go on by introducing even more arrows: $a \uparrow\uparrow\uparrow b = a(\uparrow\uparrow (b \uparrow\uparrow (b \dots)))$ and we can go on and on *ad infinitum.*

This approach gives us a lot of power to define huge numbers. But what about going backward? How can we invert the effect of power (not to mention Knuth's megapower)? There are in fact two ways to invert the power function. We may ask for the *root* a in $a^b = c$, if we know b and c, and we may ask for the *exponent* b, if a and c are known. The first operation is just called the *root*, whereas the latter is called the *logarithm* of c to base a.

Both these functions are again asymmetric in that any power of two natural numbers a^b results in a natural number, but not all natural numbers c have a natural numbered root a or a natural numbered logarithm b to base a. It is possible to define natural numbered approximations to the precise results. But, since we will not make any use of such functions, we will not implement them here. We come back to root and log algorithms later.

There are three bases whose logarithms are particularly interesting: the logarithm base 10 ($\log_{10}$) is intersting when we are working in the decimal system. The logarithm base 2 ($\log_2$) is interesting, when working with the binary system, but also in many areas where binarity plays a role, some of which we will explore later. Then there is the logarithm to the base e ($\log_e$), the so called *natural logarithm.* This number e, which is approximately 2.71828, is one of the most curious mathematical objects. It appears again and again in apparently unrelated problem areas such as number theory, calculus and statistics. It especially loves to appear, when you least expect it. We have no means to express this number with natural numbers, so we have to come back to it later to define it properly.

Unfortunately, there are different shorthands for these logarithms in different contexts. Computer scientists would write the binary logarithm log, because it is the most common in their field. This shorthand, however, usually means the natural logarithm in most math publications and even many programming languages, including Haskell, use the symbol log for $\log_e$. To make it worse, in many engineering disciplines, $\log_{10}$ is considered the most common logarithm and, accordingly, log is considered to mean $\log_{10}$. There is an ISO standard, which apparently isn't followed by anybody, that gives the following reasonableconvention: $\log_2 = lb$, $\log_e = ln$ and $\log_{10} = lg$. But even these shorthands are often confused. The best way, therefore, appears to be to use explicit symbols with subscripts.

Logarithms adhere to very interesting arithmetic rules that often reduce computational complexity in dealing with huge numbers. The logarithm (base b) of the product of two numbers equals the sum of the logarithm (base b) of these numbers: $\log_b(n \times m) = \log_b(n) + \log_b(m)$. Example: $\log_2(4 \times 8) = \log_2(32) = 5$ and $\log_2(4) + \log_2(8) = 2 + 3 = 5$.

Accordingly, the logarithm of the quotient of two numbers equals the difference of the numerator and denominator: $\log_b(\frac{n}{m}) = \log_b(n) - \log_b(m)$, for instance $\log_2(\frac{32}{8}) = \log_2(4) = 2$ and $\log_2(32) - \log_2(8) = 5 - 3 = 2$.

The logarithm of a power of a number n equals the exponent multiplied with the logarithm of n: $\log_b(n^x) = x \times \log_b(n)$, *e.g.*: $\log_2(4^3) = \log_2(64) = 6$ and $3 \times \log_2(4) = 3 \times 2 = 6$.

Finally, the logarithm of a root of n equals the logarithm of n divided by the exponent: $\log_b(\sqrt[x]{n}) = \frac{\log_b(n)}{x}$, for example: $\log_2(\sqrt[3]{64}) = \log_2(4) = 2$ and $\frac{\log_2(64)}{3} = \frac{6}{3} = 2$.

We can also convert logarithms with different bases to each other. Let us assume we want to convert the logarithm base b of a number n to the logarithm base a of n; then $\log_a n = \frac{\log_b n}{\log_b a}$, *i.e.* we divide the logarithm $\log_b n$ by the logarithm $\log_b$ of a. We will later show why this rule holds.

3.7. Numbers as Strings

Until now we have looked at numbers as sequences of symbols, *i.e. strings*. In the next section that will end. We will then define our numbers as a fully-fledged Haskell number type. But before we do that, we will pause shortly to make the difference between the two viewpoints on numbers quite clear. Indeed, in many math problems, the representation of numbers as strings is relevant – especially in informatics. So, this viewpoint is not only related to the way how we happened to define our numbers, but is a genuine mathematical approach.

We have already seen some strange effects of multiplication on the characteristics of the resulting sequences of digits. A much simpler example that shows the properties of numbers as being strings is typing errors. There is no obvious numerical analogy between number pairs like 12 and 21, 26 and 62 or 39 and 93. But, obviously, there is a very simple function that produces these numbers, namely *reverse*:

$reverse\ [One, Two] = [Two, One]$
$reverse\ [Two, Six] = [Six, Two]$
$reverse\ [Three, Nine] = [Nine, Three]$

That is not a numeric property, but a property of any kind of sequence of symbols. Numbers as such, however, are not sequences of symbols. We rather make use of sequences of symbols to represent numbers. In some way, however, any formal system used to represent numbers will have the form of sequences of symbols and, as such, numbers exist in both worlds, a *purely* numerical and a symbolic world.

There is a well known sequence of natural numbers living on the very border between the numerical and the string side of numbers, the *look-and-say* sequence, which is often used in recreational math, but is also investigated by serious (even if playful) mathematicians, such as John H. Conway, co-author of the Book of Numbers. Can you guess how to continue the following sequence?

$1, 11, 21, 1211, 111221, \dots$

The sequence starts just with one. The next number explains to us what its predecessor looks like: it is composed of one "one". This number, now, is composed of two "ones", which, in its turn, is composed of one "two" and one "one". This again is composed of one "one", one "two" and two "ones". Now, you are surely able to guess the next number.

There are some interesting questions about this sequence. What is the greatest digit that will ever occur in any number of this sequence? Well, we can easily prove that this digit is 3. The numbers of the sequence are composed of pairs of digits that describe groups of equal digits. The first digit of each pair says how often the second digit appears in this group. The number 111221, for instance, describes a number composed of three group: 11 12 21. The first group consists of one "one", the second group of one "two" and the last group of two "ones". Now, it may happen that the digit of the current group coincides with the number of digits in the next group. But the digit in that group must differ from the digits in the current group. Otherwise, it would belong to the current group. A good example is 11 12: if the forth number were 1, like 11 11, then we would have said 21 in the first place. Therefore, there will never be more than three equal numbers in a row and the greatest number to appear in any number is thus 3. □

How can we implement this sequence in Haskell? There seem to be two different principles: First, to describe a given number in terms of groups of digits and, second, to bootstrap a sequence where each number describes its predecessor. Let us implement these two principles separately. The first one is very simple:

$$\begin{array}{l} say :: Number \to Number \\ say\ xs = concat\ [len\ x +\!\!+ [head\ x] \mid x \leftarrow group\ xs] \end{array}$$

The *group* function is defined in *Data.List* and groups a list according to repeated elements, exactly what we need. On each element of its result set, we apply *len*, the *length* function for natural numbers we defined earlier, and concatenate this result with the head of that element. For instance, *group* [*One*, *Two*, *One*, *One*] would give [[*One*], [*Two*], [*One*, *One*]]. The length of the first list is [*One*] and concatenated with the head of [*One*] gives [*One*, *One*]. The length of the second list, again, is [*One*] and concatenate with the head of [*Two*] gives [*One*, *Two*]. The length of the third list is [*Two*] and concatenated with the head of [*One*, *One*] is [*Two*, *One*]. Calling *concat* on these results gives [*One*, *One*, *One*, *Two*, *Two*, *One*], which converted to an *Integer*, is 111221.

This function is more general than the sequence, however. We can apply it on any number, also on numbers we would never see in the look-and-say sequence. Applied on *unity*, *say* would just give [*One*, *One*]. Then, from *two* to [*Nine*], the results are quite boring: [*One*, *Two*], [*One*, *Three*], ..., [*One*, *Nine*]. But applied on *ten*, it would result in [*One*, *One*, *One*, *Zero*].

We will now use *say* to implement the look-and-say sequence starting from 1:

```
says :: Number → Number
says []     = []
says [Zero] = []
says [One]  = [One]
says n      = say (says (prev n))
```

First, we handle the cases that are not part of the sequence: the empty list and *zero*. Then, we handle [*One*], which is just [*One*]. Finally, we define the sequence for any number as *say* of *says* of the predecessor of that number. For instance:

```
say [Three] = say (says (prev [Three]))
say [Three] = say (says [Two])
say [Three] = say (say (says (prev [Two])))
say [Three] = say (say (says [One]))
say [Three] = say (say [One])
say [Three] = say ([One, One])
say [Three] = [Two, One].
```

Sometimes, the two sides of numbers, their numeric properties and their nature as sequences of digits, become entangled. This is the case with *narcissitic numbers*, a popular concept in recreational math – without further known applications in math or science. Narcissistic numbers are defined by the fact that they equal the sum of their digits raised to the power of the number of digits in the whole number. More formally, a narcissistic number n is a number for which holds:

$$n = \sum_{i=0}^{s} n_i^s,$$

where n_i is the digit of n at position i and s is the number of digits in n. In fact, we can define the property of being narcissistic much clearer as a test in Haskell using our number type:

$$
\begin{aligned}
&\mathit{narcissistic} :: \mathit{Number} \rightarrow \mathit{Bool}\\
&\mathit{narcissistic}\ n = \mathit{foldr}\ (\mathit{step}\ (\mathit{len}\ n))\ \mathit{zero}\ n \equiv n\\
&\quad \textbf{where}\ \mathit{step}\ s\ a\ b = b\ \text{`}\mathit{add2}\text{`}\ (\mathit{power2}\ [a]\ s)
\end{aligned}
$$

This property holds trivially for all numbers $< \mathit{ten}$. Then, they get rare. The narcissistic numbers between 10 and 1000 are: 153, 370, 371 and 407. 153, for instance, is narcissistic because $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$. Interesting is the pair 370 and 371: $370 = 3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370$. Now, if we add 1, *i.e.* $1^3 = 1$, 371 arises.

The number of narcissistic numbers in a given number system is limited. This is because for sufficient large ks, the smallest possible number of the form 10^{k-1}, *i.e.* the smallest number with k digits, is greater than the greatest number of the form $k \times 9^k$, *i.e.* the greatest number we can build by adding up the k^{th} powers of the digits of a k-digit-number. That means that, for large numbers, the numerical value will always be greater than the sum of the digits raised to the number of digits in that number. In the decimal system, this limit is reached with $k = 61$. 10^{60} is obviously the smallest number with 61 digits. The 61-digit number with which we can build the greatest sum of 61^{st} powers is the number $99\dots9$ that consists of 61 9s. If we raise all these 9s to the 61^{st} power and sum the results, we will obtain a number with 60 digits. That number is clearly less than the least number we can represent with 61 digits. Therefore, no narcissistic numbers with more than 60 digits are possible. In practice, there are only 88 narcissistic numbers in the decimal number system and the greatest of those has 39 digits.

Another popular problem from recreational math is that of a 10-digit number, where each position tells how often the digit related to that position counted from left to right and from 0 to 9 is present in the number. If we represent such a number as in the following table

0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i	j

then a would tell how often 0 appears in that number, b, how often 1 appears in that number, c, how often 2 appears in that number and so on.

How can we tackle that problem? First, obviously, the numbers we write in the second line of the table must add up to 10, since these numbers tell how often the related digit appears in the whole number. Since the number has 10 digits, there must be in total 10 occurrences.

We can further assume that the most frequent digit that appears in the number is 0. Otherwise, if a greater digit appeared with high frequency, it would imply that also other numbers must appear more often, since every digit that appears in the number implies another digit to appear. For instance, if 5 was the number with most occurrences, then some numbers must appear 5 times, namely those where we actually put the number 5.

So, let us just try. We could say that 0 occurs 9 times. We would have something like

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

This means that 0 appears 9 times and 9 appears once. But there are two problems with this solution: First, if 9 appears once, then 1 appears once as well, but, then, there are only 8 places left to put 0s in. Second, if 1 appearas once (to count 9), then we must put 1 below 1 in the table. But then 1 appears twice, so we must put 2 below 1 and, as a consequence, we must put 1 below 2. In fact, whatever the number of 0s is, for all solutions, we need at least two 1s and one 2, hence:

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

Let us think: We have to convert two of the xs into numbers, one into a number that we do not know yet and the other to 1 to count that unknown number. In other words, we will have 2, 1, 1 and some other number. Since we know that the numbers must add up to 10, we can just compute that unknown number as $x = 10 - (2 + 1 + 1) = 10 - 4 = 6$. The result then is 6 210 001 000:

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

Is this the only possible configuration, or are there others that fulfil the constraints? Let us assume there is another configuration. We already know that 7,8 and 9 do not work. So, instead of 6, we will have a number smaller than 6. This number could be 5. Then, we have to make up the difference between 6 and 5, since the numbers, at the end, must add up to 10. That means, we need one more 1. But, then, we need an additional number that occurs once to justify that additional 1. But, since there is only room for one additional number, that cannot be.

Then, we could try 4. But 4 does not work either, since the difference now is 2 and we cannot just increase the ocurrences of 1 or 2 without justification. If we increased the occurrences of 1, we would have to add another number, to justify that additional 1. We need, in fact, three numbers, but we have only room for two.

Then 3 could be a solution: instead of one 6, we would have two 3s. But now, we must justify the second 3 and there is no room for another number appearing three times. So, since 1 and 2, obviously, will not work, we conclude, that 6 210 001 000 is the only possible configuration.

We are now very close to leave the world where we look at numbers mainly as strings. We will soon look at numbers in a completely different way. But before we do that, we still have to finalise the model of our number type, that is, we should define how to convert an integer into our *Number*. We can of course just convert the integer into a string using *show* and then convert the string digit by digit to *Number*. But, again, that would be boring. We would not learn anything special about numbers, which is the main concern of all our exercises here.

Instead, we will think along the lines of decimal numbers being representations of powers of 10. We will ask: how many powers of 10 does a given number contain? How many powers of 10 are, for example, in the number 9827? To answer this question, we first have to find the floor of $\log_{10} 9827$, *i.e.* a number l such that $10^l \leq 9827$ and $10^{l+1} > 9827$: $\lfloor \log_{10} 9827 \rfloor$. For 9827, that is 3, since $10^3 = 1000$ and $10^4 = 10000$. To learn how many third powers of 10 are in the number, we divide the number by the third power of 10: $\lfloor \frac{9827}{10^3} \rfloor = 9$. We, hence, have 9 times the third power of 10 in 9827. The first digit is therefore *Nine*. To convert the whole number, we now apply the algorithm on the remainder of the division $\frac{9827}{10^3}$, which is 827.

But hold on: is this not quite expensive with a log operation and a division on each digit of the original integer? Yes, in fact, we can think much simpler in terms of modulo. A number in the decimal system is composed of the digits $0 \dots 9$. Any number modulo 10 is one of these digits. The remainder of 9827 and 10, for instance, is 7, because the Euclidian division of 9827 and 10 is $(982, 7)$; the Euclidian division of 982 and 10 is $(98, 2)$; the result for 98 and 10 is $(9, 8)$ and that for 9 and 10 is just $(0, 9)$. In other words, we can just collect the remainders of the Euclidian division of the integer and 10 and convert each digit into our *Digit* type. Here is the code in Haskell:

```
integer2Num :: Integer → Number
integer2Num 0    = zero
integer2Num 1    = unity
integer2Num 2    = two
integer2Num 3    = [Three]
integer2Num 4    = [Four]
integer2Num 5    = [Five]
integer2Num 6    = [Six]
integer2Num 7    = [Seven]
integer2Num 8    = [Eight]
integer2Num 9    = [Nine]
integer2Num 10   = ten
integer2Num i    = go i
  where go n = case n `quotRem` 10 of
                 (0, r) →           integer2Num r
                 (q, r) → go q ⧺ integer2Num r
```

We start by handling all one-digit numbers and 10 explicitly This has two advantages: we speed up the processing for one-digit numbers and 10 and we do not need an extra conversion function for digits.

For all values of i not handled in the base cases, we compute quotient and remainder. If the quotient is 0, we are done with *go* and just yield the conversion of r, which must be a digit, since it is a remainder of division by 10. Otherwise, we continue with the quotient to which we append the conversion of the remainder.

For the example 9827, we would create the following sequence:

$$\begin{aligned}
&go\ 9827 = go\ 827 +\!\!+\ integer2Num\ 7\\
&go\ 982 = go\ 82 +\!\!+\ integer2Num\ 2 +\!\!+\ integer2Num\ 7\\
&go\ 98 = go\ 8 +\!\!+\ integer2Num\ 8 +\!\!+\ integer2Num\ 2 +\!\!+\ integer2Num\ 7\\
&go\ 9 = integer2Num\ 9 +\!\!+\ integer2Num\ 8 +\!\!+\ integer2Num\ 2 +\!\!+\ integer2Num\ 7,
\end{aligned}$$

which is

$$\begin{aligned}
&[Nine] +\!\!+ [Eight] +\!\!+ [Two] +\!\!+ [Seven]\\
&[Nine, Eight, Two, Seven].
\end{aligned}$$

3.8. $\mathbb{N}$

We will now convert our *Number* type in a full-fledged Haskell *Num* type. This will allow us to use numeric symbols, *i.e.* the number $0 \ldots 9$, instead of the constructors *Zero*... *Nine*, for our type and we will be able to use the standard operators $+, -, *$. The first step is to define a **data** type – until now, we used only a type synonym for [*Digit*]:

```
data Natural = N Number
```

The new data type is called *Natural* and its only constructor is N receiving a *Number*, *i.e.* [*Digit*], as parameter. The constructor N is named after the symbol for the set of natural numbers in math, which is $\mathbb{N}$.

We, then, make this data type instance of *Eq* and *Show*, where we use the previous defined functions *cmp* for comparisons and *n2Integer* for conversion to *Int* and subsequent *show*:

```
instance Eq Natural where
  (N a) ≡ (N b) = cmp a b ≡ EQ
instance Show Natural where
  show (N ns) = show (n2Integer ns)
```

Now we are ready to make *Natural* instance of *Num*. *Num* has the following methods we have to implement: $+$, $-$, $*$, *negate*, this would be a negative number, which we have not yet defined, so we leave this method undefined, *abs*, the absolute value of a number, *signum*, which is either 0 (for *zero*), 1 (for numbers > 0) or -1 (for numbers < 0), and *fromInteger*, a conversion function that turns instances of type class *Integral*, like *Int* and *Integer*, into our data type. Here is the code:

```
instance Num Natural where
  (N as) + (N bs)                      = N (as `add2` bs)
  (N as) − (N bs) | cmp as bs ≡ LT = error "subtraction below zero"
      | otherwise                      = N (as `sub2` bs)
  (N as) ∗ (N bs)                      = N (as `mul2` bs)
  negate  n                            = ⊥
  abs     n                            = n
  signum (N [Zero])                    = 0
  signum n                             = 1
  fromInteger i                        = N (integer2Num i)
```

Two *Natural*s are added by adding the *Number*s of which they consists using *add2* and calling the constructor *N* on the result. Subtraction and multiplication are implemented accordingly using *sub2* and *mul2* respectively. *abs n* is just *n*, since Natural is always a positive number, we do not need to worry about negative numbers passed in to *abs*. *signum* for *zero* is just 0, for any other number, it is 1. Again, because of their absence, we do not need to handle negative numbers. For *fromInteger*, we finally use the conversion function *integer2Num*.

There are some other properties we would like our number type to have. First, numbers, in Haskell, are also *Enum*s, *i.e.* objects that can be enumerated. The class *Enum* defines the methods *succ* and *pred* – which we already know from Peano numbers – *toEnum*, the conversion of integrals, especially *Int*s, to our data type, and *fromEnum*, the opposite conversion:

```
instance Enum Natural where
  succ (N n)         = N (next n)
  pred (N [Zero])    = error "zero has no predecessor"
  pred (N n)         = N (prev n)
  toEnum             = N ∘ integer2Num ∘ fromIntegral
  fromEnum (N n) = fromIntegral (n2Integer n)
```

Numbers, additionally, have order. For every two numbers, we can say which of the two is greater or less than the other. This is captured by the type class *Ord*. The only method we have to implement for making *Natural* instance of *Ord* is compare:

```
instance Ord Natural where
  compare (N as) (N bs) = cmp as bs
```

We also want to be able to convert our numbers into real numbers. To do so, we make *Natural* an instance of *Real*:

```
instance Real Natural where
```

```
    toRational (N ns) = fromIntegral $ n2Integer ns
```

Finally, our number type is a kind of integral, *i.e.* not a fraction. To express this in Haskell, we make *Natural* instance of the *Integral* class and implement the methods *quotRem* and *toInteger*. The code is quite obvious, no further explanations are necessary:

```
  instance Integral Natural where
    quotRem (N as) (N bs) = let (q, r) = D.quotRem2 as bs in (N q, N r)
    div        a b        = fst $ quotRem a b
    toInteger (N ns)      = n2Integer ns
```

3.9. Abstract Algebra

When we look at the four fundamental arithmetic operations, addition, multiplication, subtraction and division, we see some striking differences between them. We can, more specifically, distinguish two groups of operations, namely addition and multiplication on one side and subtraction and division on the other.

for multiplication and addition, we can state that for any two natural numbers a and b, the result of the operations $a + b$ and $a \times b$ is again a natural number. For subtraction and division that is not true. As you may remember, for subtraction, we had to define an important exception, *viz.* that the second term must not be greater than the first one. Otherwise, the result is not a natural number.

Division according to Euclid, besides having the exception of *zero* in the denominator, differs completely, in that its result is not at all a number, but a pair of numbers (q, r). If we refer to division in terms of the *quot* operation (which returns only the quotient, not the remainder), then division would indeed behave similar to addition and multiplication (besides the division-by-zero exception). But that would leave us with a torso operation that falls behind the other operations in precision and universality.

Another property shared by addition and multiplication is the *associative law*:

$$a + (b + c) = (a + b) + c = a + b + c \tag{3.5}$$

$$a \times (b \times c) = (a \times b) \times c = a \times b \times c \tag{3.6}$$

This, again, is not true for subtraction and division, as can be easily shown by counter examples:

$$4 - (3 - 1) \neq (4 - 3) - 1,$$

since

$$4 - (3 - 1) = 4 - 2 = 2$$

and

$$(4 - 3) - 1 = 1 - 1 = 0.$$

For division, we cannot even state such an equality (or inequality), since the result of the Euclidian division, a pair of numbers, cannot serve as one of its arguments, *i.e.* a pair of numbers cannot be divided. If we, again, accept the *quot* operation as a compromise, we quickly find counter examples which show that the associative law does not hold for division either:

$$3/(2/2) \neq (3/2)/2,$$

since

$$3/(2/2) = 3/1 = 3$$

and

$$(3/2)/2 = 1/2 = 0.$$

Abstract Algebra uses such properties to define different classes of numbers and other "things" – of which we will soon see some examples. The first class of such things we can define is the *magma* or *groupoid*. A magma is a set together with a binary operation such that the set is closed under this operation. We will look at sets in more detail in the next section; for the moment, we can live with an informal intuition of sets being collections of things, here of certain types of numbers. That a set is closed under an operation is just the property we defined first, *i.e.* that c is a natural number if a and b are natural numbers in $a + b = c$. More formally, we can describe a magma as

$$M = (S, \cdot), \tag{3.7}$$

where S is a set and $\cdot$ is a binary operation, such that for all $a, b \in S$ (a and b are *element of* S, *i.e.* they are members of the set S), $a \cdot b \in S$, *i.e.* the result of the operation $a \cdot b$, too, is in S.

When we add the other property, the associative law, to the magma definition, we get a *semigroup.* A semigroup, hence, is a magma, where for the operation $\cdot$ the relation $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ holds.

Natural numbers with either addition or multiplication are clearly semigroups. We can be even more specific: Natural numbers are *abelian semigroups*, since the *commutative law* holds for them as well. The commutative law states that, for an operation $\cdot$ the relation: $a \cdot b = b \cdot a$ holds, which, again, is not true for subtraction and division.

The next property is the identity. This property states that there is an element e in S, for which holds that $a \cdot e = e \cdot a = a$. For addition and subtraction, this element e is *zero*. For multiplication, it is *unity*. For a division operation defined as *quot* this element would be *unity* as well.

A semigroup with identity is called a *monoid.* Natural numbers are hence abelian monoids, since the commutative law holds for them as well. An example for a non-abelian monoid is the set of all strings, STR, with the concatenation operation $+\!\!+$. First note that STR is closed under concatenation, since, for any strings a and b, it holds that (using Haskell syntax) $a +\!\!+ b$ is again a string, *e.g.* `"hello "` $+\!\!+$ `"world"` $\equiv$ `"hello world"`.

Then, the associative law holds, since for any three strings, a, b and c: $a +\!\!+ (b +\!\!+ c) \equiv (a +\!\!+ b) +\!\!+ c \equiv a +\!\!+ b +\!\!+ c$, for instance: `"hello"` $+\!\!+$ (`" "` $+\!\!+$ `"world"`) $\equiv$ (`"hello"` $+\!\!+$ `" "`) $+\!\!+$ `"world"` $\equiv$ `"hello world"`.

Next, there is an identity, *viz.* the empty string "", such that: $a +\!\!+$ `""` $\equiv$ `""` $+\!\!+ a \equiv a$, for instance: `"hello world"` $+\!\!+$ `""` $\equiv$ `"hello world"`.

Note, however, that the STR monoid is not commutative: $a +\!\!+ b \not\equiv b +\!\!+ a$, for instance: `"hello "` $+\!\!+$ `"world"` $\not\equiv$ `"world"` $+\!\!+$ `"hello "`.

The term *semigroup* suggests that there is also something called a *group*, which, in some way, is more complete than a semigroup – and, indeed, there is. A group is a monoid with the addtional property of *invertibility.* Invertibility means that there is an element to invert the effect of an operation, such that for any a and b for which holds: $a \cdot b = c$, there is an element x, such that: $c \cdot x = a$. Note that this implies for b and its inverse element x: $b \cdot x = e$, where e is the identity.

Unfortunately for our poor natural numbers, there are no such elements with addition and multiplication. Note, however, that, if we had already introduced negative numbers and fractions, there would ineed exist such elements, namely for addition: $a + x = 0$, where obviously $x = -a$ and for multiplication: $a \times x = 1$ with $x = \frac{1}{a}$.

Let us summarise the fundamental properties of binary operations to keep track of all the properties that may hold for different types of objects:

	closure	associativty	identity	invertibility	commutativty
	$a \cdot b \in S$	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$	$a \cdot e = e \cdot a = a$	$a \cdot \frac{1}{a} = e,$	$a \cdot b = b \cdot a$
magma	×				
semigroup	×	×			
monoid	×	×	×		
group	×	×	×	×	
abelian x	×	-	-	-	×

It is to be noted that any of the concepts magma, semigroup, monoid and group may have the property of being abelian. There are abelian magmas, semigroups, monoids and groups. Therefore, the *abelian x* is indifferent towards associativity, identitiy and invertibility. This depends entirely on the x, not on the x being abelian or not.

We will now introduce a major step. We have, so far, added additional propterties to magmas, semigroups and so on to create new kinds of objects. Now, we change the underlying definition to create something completely different, namely a *semiring*. A semiring is a set S together with **two** binary operations, denoted $\bullet$ and $\circ$:

$$R = (S, \bullet, \circ). \tag{3.8}$$

The operation $\bullet$ must form an abelian monoid with S and the operation $\circ$ must form a monoid (which may or may not be abelian) with S. These conditions are fulfilled for addition and multiplication on the natural numbers. Since both, addition and multiplication, form abelian monoids, for the moment, both may take either place in the definition. But there is one more property: the operations together must adhere to the *distributive law*, which states that

$$a \circ (b \bullet c) = (a \circ b) \bullet (a \circ c). \tag{3.9}$$

This, again, is true for natural numbers, if $\bullet$ corresponds to addition and $\circ$ to multiplication: $a \times (b + c) = (a \times b) + (a \times c)$. We can simplify this formula by leaving the parentheses out of course: $a \times b + a \times c$ and can even further simplify by adopting the usual convention that $a \times b = ab$: $ab + ac$.

A *ring* is a semiring, for which the additional property of invertibility holds on addition. A ring, hence, consists of an abelian group (addition in case of natural numbers) and a monoid (multiplication). Again, natural numbers do not form a ring, but only a semiring, since there are no negative numbers in natural numbers and there is thus no inverse for addition.

A ring, where multiplication is commutative, hence, a ring with an abelian group (addition) and an abelian monoid (multiplication) is called a *commutative ring*.

The most complete structure, however, is the *field*, where both operations, addition and multiplication are abelian groups.

Here is the complete taxonomy:

	addition	multiplication
semiring	abelian monoid	monoid
ring	abelian group	monoid
commutative ring	abelian group	abelian monid
field	abelian group	abelian group

4. Induction, Series, Sets and Combinatorics

4.1. Logistics

Before we continue to investigate the properties of natural numbers, let us deviate from pure theory for a moment and have a look at a motivating example from my professional practice. It is quite a simple case, but, for me, it was one of the starting points to get involved with sequences, series, combinatorics and other things natural numbers can do.

I was working for a logistics service provider for media, mainly CDs, DVDs, video games, books and so on. The company did all the merchandise management for its customers, mainly retailers, and, for this purpose, ran a set of logistics centres. We got involved, when one of those logistics centres was completely renewed, in particular a new sorter system was installed that enabled the company to comfortably serve all their current customers and those expected according to steep growth rates in the near future.

A sorter is a machine that reorders things. Goods enter the warehouse ordered by the suppliers that actually sent the goods in lots of, for instance, 1.000 boxes of album A + 450 boxes of album B + 150 boxes of album C. These lots would go onto the sorter and the sorter would reorder them into lots according to customer orders, *e.g.*: customer I ordered: 150 boxes of album A + 30 boxes of album B + 10 boxes of album C, customer II ordered: 45 boxes of album A + 99 boxes of album B and so on.

Mechanically, the sorter consisted of a huge belt with carrier bins attached to it that went around in circles. Feeders would push goods onto the carrier bins and, at certain positions, the bins would drop goods into buckets on the floor beneath the belt, so called endpoints. At any time, endpoints were assigned to customers, so that each endpoint ended up with goods ordered by one specific customer.

Our software was responsible for the configuration of the machine. It decided, which customers were assigned to endpoints and how goods were related to customer orders. The really tricky task was optimising the process, but here I would like to focus on one single issue that, in fact, was much simpler than all that optimisation stuff, namely the allocation of customers to endpoints.

At any given time, the sorter had a certain allocation, that is an assignment of endpoints to customers. There were very big customers that received several lots per day and others that would only receive lots on certain weekdays. Only those customers that would still receive a lot on the same day and within the current batch would actually have an allocation. The goods for those, currently not on the sorter, would fall in reserved endpoints, called “ragmen”, for later batches or other weekdays. With this logic, the sorter was able to serve much more customers than it had endpoints, and what we wanted to know was how many ragmen we would need with respect to a given amount of customers.

Our idea for attacking the problem was the following: we started to assume naïvely that we could simply split the customers by some ratio in those currently *on* (assigned to an endpoint) and those currently *off* (not assigned to an endpoint). We would split, let us say, 1.000 customers into 500 allocated to some endpoint and 500 currently not allocated. But, unfortunately, we needed some endpoints to catch all the merchandise intended for those currently not *on*. So, we had to reserve a certain amount of endpoints as ragmen and subtract this number from the amount of endpoints available for allocated customers. A key variable was the number of customers *off* per ragman endpoint. We wanted this number, of course, to be as high as possible, because from this relation came the saving in endpoints that must be reserved as ragmen and it finally determined, how many customers the server could serve. On the other hand, we could not throw the goods for all customers currently not at the sorter into one single endpoint. This would have caused this endpoint to overflow every few minutes causing permant work in getting the merchandise to a waiting zone. This special point turned out to be quite complicated: small customers with small lots would need less ragman capacity than big ones; the problem was solved with a classification approach, but that does not matter here at all. For our purpose, it is just important that there actually was some value to determine this relation, let us say it was $c = 10$, meaning that we needed a ragman endpoint for every 10 customers not on the sorter.

We will now use the naïve assumption to compute the number of ragmen as $r = \lceil \frac{n-m}{c} \rceil$, where n is the number of customers and m the number of available endpoints. For our example of 1.000 customers and 500 endpoints, r is $\frac{1000-500}{10}$, hence, 50 ragman endpoints.

But this result cannot be true! We naïvley assumed that we have 500 endpoints. But in the very moment we reserve 50 endpoints as ragmen for customers not currently on the sorter, this number reduces instantly to $m - r$, that is 450 endpoints. We, therefore, have to reserve more ragmen, that is to say for those 50 customers that, now, have no endpoint on the sorter anymore. Since we need one ragman per 10 customers, this would give $50 + 5$ ragmen. But would this not reproduce the problem we wanted to solve in the first place? In the very moment, we add 5 more endpoints to the ragmen, we have to take away 5 from the available endpoints, reducing the number of available endpoints once again to $450 - 5 = 445$.

We end up with something called a series: the number of ragmen equals the number of endpoints divided by c plus this number divided by c plus this number divided by c and so on. We can represent this with a nice formula as:

$$r = \left\lceil \frac{n-m}{c} \right\rceil + \left\lceil \frac{n-m}{c^2} \right\rceil + \dots \tag{4.1}$$

Or even nicer:

$$r = \sum_{k=1}^{\infty} \left\lceil \frac{n-m}{c^k} \right\rceil \tag{4.2}$$

You can easily convince yourself that dividing $n - m$ by c^2 is the same as dividing $\frac{n-m}{c}$ by c, because dividing a fraction by a natural number is equivalent to multiplying it with the denominator (we will look at this more carefully later). In the sum in equation 4.2, the k is therefore growing with each step.

But the equation, still, has a flaw. The inner division in the summation formula will leave smaller and smaller values that, at some point, become infinitesimally small. but, since we ceil the division result, these tiny values will always be rounded up to one, such that the formula produces an endless tail of ones, which is of course not what we want. Therefore, we should use the opposite of ceiling, floor, but should not forget to add one additional ragman to cope with the remainders:

$$r = 1 + \sum_{k=1}^{\infty} \left\lfloor \frac{n-m}{c^k} \right\rfloor \tag{4.3}$$

Now, when $\frac{n-m}{c^k}$ becomes less than one, the division result is rounded down to zero and the overall result of the summation converges to some integer value. For 1000 customers, the series converges already for $k = 3$; we, thus, need $50+5+1 = 56$ ragmen to cope with 1000 customers and will be able to serve 444 customers on the sorter. For, say, 2.000 customers, the series converges for $k = 4$, so we need $\lfloor \frac{1500}{10} \rfloor + \lfloor \frac{1500}{100} \rfloor + \lfloor \frac{1500}{1000} \rfloor + 1 = 167$ ragmen and will have 333 endpoints *on*. For 5.000 customers, the series, again, converges for $k = 4$ and we will need $\lfloor \frac{4500}{10} \rfloor + \lfloor \frac{4500}{100} \rfloor + \lfloor \frac{4500}{1000} \rfloor + 1 = 500$, which is just the amount of endpoints we have available in total. We, thus, cannot serve 5.000 customers with this configuration. We would need to increase c and accept more workload in moving goods into wating zones.

Let us look at a possible implementation of the above with our natural numbers. First, the notion of *convergence*, as we have used it above, appears to be interesting enough to define a function for it. The idea is that we sum up the results of a function applied to an increasing value until the result reaches zero and, in consequence, will not affect the cumulated result anymore:

```
converge1 :: (Natural → Natural) → Natural → Natural
converge1 f n = let r = f n
```

```
                          in if r ≡ 0 then r
                                      else r + converge1 l f (n + 1)
```

The function *converge* receives a function f that transforms a natural number into another natural number and the natural number n, which is the starting point for the series. We compute the result r of $f\ n$ and if this result equals zero, we produce the result r, otherwise, we continue with $n + 1$.

We can generalise this function so that it is also applicable to products. In this case, we would not stop, when f produces 0, but when it produces 1, the neutral element with respect to multiplication. The definition of the generalised convergence function must hence include the stop signal explicitly as one of its arguments:

```
converge :: Natural → (Natural → Natural  → Natural) →
                      (Natural → Natural) → Natural  → Natural
converge l con f n = let r = f n
                     in if r ≡ l then r
                                 else r ‘con‘ converge l f (n + 1)
```

This version is very similar to the previous one, but it accepts two more arguments: The first argument, l, is the neutral element with respect to the combination function, *con*, passed in as the second argument (*Natural* → *Natural* → *Natural*), *i.e.* addition or multiplication. The implementation of the function differs in only two aspects: We compare the result not explicitly with 0, but with l, the limit passed in, and, instead of (+), we use ‘*con*‘ to combine results.

From here, we can very simply define two derived functions *convSum* and *convProduct*:

```
convSum :: (Natural → Natural) → Natural → Natural
convSum = converge 0 (+)
convProduct :: (Natural → Natural) → Natural → Natural
convProduct = converge 1 (∗)
```

Let us look at how to use the convergence function:

```
ragmen :: Natural → Natural → Natural → Natural
ragmen n m c = 1 + convSum (f n m c) 1
  where f :: Natural → Natural → Natural → Natural → Natural
        f n m c k = (n − m) ‘floorDiv‘ (c ↑ k)
```

The *ragmen* function simply adds one to the result of a call to the *convSum* function defined above. The function f passed to *convSum* and defined in the **where** clause can be easily recocnised as the ragman function defined in the text above. We pass f with n, m and c, thas is the number of customers, the number of endpoints and the constant c to *convSum*. We additionally pass 1 as the first value of k.

4.2. Induction

The series we looked at in the previous section converge very soon, for realistic values, after 3 or 4 steps. But this may be different and then huge sums would arise that are costly to compute, since many, perhaps unfeasibly many additions had to be made. We already stumbled on such problems, when we looked at multiplication. It is therefore often desirable to find a *closed form* that leads to the same result without the necessity to go through all the single steps. Let us look at a very simple example. We could be interested in the value of the n first odd numbers summed up, *i.e.* for $n = 2$: $1 + 3 = 4$, $n = 3$: $1 + 3 + 5 = 9$, $n = 4$: $1 + 3 + 5 + 7 = 16$ and so on. With large values of n, we would have to go through many steps, *viz.* $n - 1$ additions.

First, let us think about how to express this as a formula. An odd number is a number that is not divisible by 2. Even numbers could be expressed as $2k$ for all ks from $1 \dots n$, for instance the first even number, $n = 1$, is 2, the first two even numbers, $n = 2$, are 2 and 4, since $2 \times 2 = 4$, the first three even numbers, $n = 3$, are 2, 4 and 6, since $2 \times 3 = 6$ and so on. Odd numbers, correspondingly, can be described as: $2k - 1$. The first odd number, hence, is $2 \times 1 - 1 = 1$, the first two odd numbers, $n = 2$, are 1 and 3, since $2 \times 2 - 1 = 3$, the first three odd numbers, $n = 3$, are 1, 3 and 5, since $2 \times 3 - 1 = 5$ and so on. Correspondingly, the sum of the first n odd numbers can be properly described as:

$$\sum_{k=1}^{n}(2k - 1)$$

To convince ourselves that this formula is correct, let us go through some examples: If $n = 1$, then $2k - 1$ equals 1, for $n = 2$, this is the result of $n = 1$ plus $4 - 1$, hence $1 + 3 = 4$, for $n = 3$, the formula leads to $4 + 6 - 1 = 9$ and for $n = 4$, the result is $9 + 8 - 1 = 16$. The formula appears to be correct.

We can implement this formula literally by a simple Haskell program:

```haskell
oddSum1 :: Natural → Natural
oddSum1 n = go 1
  where go k | k > n     = 0
             | otherwise = (2 ∗ k − 1) + go (k + 1)
```

Now, is there a closed form that spares us from going through all the additions in the *go* function? When we look at the results of the first 9 numbers calling oddSum1 as

```
map oddSum1 [1..9]
```

we see that all the numbers are perfect squares: $1, 4, 9, 16, 25, 36, 49, 65, 81$. Indeed, the results suggest that the sum of the first n odd numbers equals n^2. But is this always true or does it hold only for the first nine numbers we just happened to look at? Let us try a proof by *induction*.

Induction is an tremendously important technique, since it enables us to prove that a property holds for infinitely many numbers! A proof by induction proves that a property P holds for a base case and, by advancing from the base case to the following number, that it holds for all numbers we are interested in. Formally, we prove, for example, that $P(n) \to P(n+1)$, where $+1$ is a very common way to advance. With $+$, we actually prove that P holds for all $x > n$. But we can use induction also for with functions that advance at a different pace, for instance, we might want to prove that some property holds for even numbers, we would then advance with $+2$.

Proofs by induction consist of two parts: First, the proof that the property is true for the base case and, second, that it is still true when advancing from a number, for which we know that it is true, like the base case, to the next number.

For the example of the sum of the odd numbers, the base case, $n = 1$, is trivially true, since 1^2 and $\sum_{k=1}^{n}(2k-1)$ are both 1. Now, if we assume that, for a number n, it is true that the sum of the first n odd numbers is n^2, we have to show that this is also true for the next number $n+1$ or, more formally, that

$$\sum_{k=1}^{n+1}(2k-1) = (n+1)^2. \tag{4.4}$$

We can decompose the sum on the left side of the equal sign by taking the induction step $(n+1)$ out and get the following equation:

$$\sum_{k=1}^{n}(2k-1) + 2(n+1) - 1 = (n+1)^2. \tag{4.5}$$

Note that the part broken out of the sum corresponds exactly to the formula within the sum for the case that $k = n+1$. Since we already now that the first part is n^2, we can simplify the expression on the left side of the equal sign to $n^2 + 2(n+1) - 1$, which, again simplified, gives:

$$n^2 + 2n + 1 = (n+1)^2 \quad \square \tag{4.6}$$

and, thus, concludes the proof. If you do not see that both sides are equal, multiply the right side out as $(n+1)(n+1)$, where $n \times n = \mathbf{n^2}$, $n \times 1 = \mathbf{n}$, $1 \times n = \mathbf{n}$ and $1 \times 1 = \mathbf{1}$. Summing this up gives $n^2 + 2n + 1$.

The *oddSum* function can thus be implemented in much more efficient way:

```
oddSum :: Natural → Natural
oddSum = (↑2)
```

For another example, let us look at even numbers. Formally, the sum of the first n even numbers corresponds to: $\sum_{k=1}^{n} 2k$. This is easily implemented in Haskell as

```
evenSum1 :: Natural → Natural
evenSum1 n = go 1
  where go k | k > n     = 0
             | otherwise = 2 ∗ k + go (k + 1)
```

Applying *evenSum*1 to the test set [1..9] gives the sequence: $2, 6, 12, 20, 30, 42, 56, 72, 90$. These are obviously no perfect squares and, compared to the odd numbers $(1, 4, 9, 16, \dots)$, the results are slightly greater. How much greater are they? For $n = 1$, *oddSum* is 1, *evenSum* is 2, *evenSum* is hence *oddSum* + 1 for this case; for $n = 2$, the difference between the results 4 and 6 is 2; for $n = 3$, the difference between 9 and 12 is 3. This suggests a pattern: the difference between *oddSum* and *evenSum* is exactly n. This would suggest the closed form $n^2 + n$ or, which is the same, $n(n + 1)$. Can we prove this by induction?

For the base case $n = 1$, $\sum_{k=1}^{1} 2k$ and $n(n + 1)$ are both 2. Now assume that for some n, $\sum_{k=1}^{n} 2k = n(n + 1)$ holds, as we have just seen for the base case $n = 1$, then we have to show that

$$\sum_{k=1}^{n+1} 2k = (n + 1)(n + 2). \tag{4.7}$$

Again, we decompose the sum on the left side of the equal sign:

$$\sum_{k=1}^{n} (2k) + 2(n + 1) = (n + 1)(n + 2). \tag{4.8}$$

According to our assumption, the summation now equals $n(n + 1)$:

$$n(n + 1) + 2(n + 1) = (n + 1)(n + 2). \tag{4.9}$$

The left side of the equation can be further simplified in two steps, first, to $n^2 + n + 2n + 2$ and, second, to $n^2 + 3n + 2$, which concludes the proof:

$$n^2 + 3n + 2 = (n + 1)(n + 2) \quad \square \tag{4.10}$$

If you do not see the equality, just multiply $(n+1)(n+2)$ out: $n \times n = \mathbf{n^2}$, $n \times 2 = \mathbf{2n}$; $1 \times n = \mathbf{n}$, $1 \times 2 = \mathbf{2}$; adding all this up gives $n^2 + 2n + n + 2 = n^2 + 3n + 2$.

We can now define an efficient version of *evenSum*:

$$\begin{aligned}&\mathit{evenSum} :: \mathit{Natural} \to \mathit{Natural}\\&\mathit{evenSum}\ n = n \uparrow 2 + n\end{aligned}$$

Now, of course, the question arises to what number the first n of both kinds of numbers, even and odd, sum up. One might think that this must be something like the sum of odd and even for n, but that is not true. Note that the sum of the first n either odd or even numbers is in fact number greater than the first n numbers, since, when we leave out every second number, then the result of counting n numbers is much higher than counting all numbers, *e.g.* for $n = 3$, the odd numbers are $1, 3, 5$ and the even are $2, 4, 6$. The first 3 numbers, however, are $1, 2, 3$.

The answer jumps into the eye when we look at the formula for the sum of even numbers: $\sum_{k=1}^{n} 2k$. This formula implies that, for each n, we take twice n. The sum of all numbers, in consequence, should be the half of the sum of the even, *i.e.* $\sum_{k=1}^{n} (k) = \frac{n(n+1)}{2}$, a formula that is sometimes humorously called *The Little Gauss*.

Once again, we prove by induction. The base case, $n = 1$, is trivially true: $\sum_{k=1}^{1} k = 1$ and $\frac{1*(1+1)}{2} = \frac{2}{2} = 1$. Now assume that $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ holds for n; then, we have to prove that

$$\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}. \tag{4.11}$$

As in our previous exercises, we take the induction step out of the summation formula and get $\sum_{k=1}^{n} (k) + (n+1)$. According to our assumption, we can reformulate this as $\frac{n(n+1)}{2} + (n+1)$. We have not yet discussed how to add fractions; to do this, we have to present both values as fractions with the same denominator, which is 2. To maintain the value of $n+1$, when we divide it by 2, we have to multiply it with 2 at the same time, yielding the fraction $\frac{2(n+1)}{2} = \frac{2n+2}{2}$:

$$\frac{n(n+1)}{2} + \frac{2n+2}{2} = \frac{(n+1)(n+2)}{2} \tag{4.12}$$

After multiplying the numerator of the first fraction on the left side of the equation out $(n^2 + n)$ and then adding the two numerators we obtain, in the numerators, the formula we already know from the even numbers:

$$\frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2} \quad \square \tag{4.13}$$

The sum of the first n natural numbers in Haskell, hence is:

$$
\begin{aligned}
&natSum :: Natural \to Natural\\
&natSum = (\text{`}div\text{`}2) \circ evenSum
\end{aligned}
$$

4.3. Arithmetic and Geometric Series

4.4. The Fibonacci Sequence

We have already discussed and analysed the run time behaviour of gcd. Let us look at an intriguing example, the gcd of, say, 89 and 55. As a reminder here the definition of gcd once again:

$$
\begin{aligned}
&gcd :: Natural \to Natural \to Natural\\
&gcd\ a\ 0 = a\\
&gcd\ a\ b = gcd\ b\ (a\ \text{`}rem\text{`}\ b)
\end{aligned}
$$

We start with gcd 89 55, which is gcd 55 (89 `*rem*` 55) after one step. What is the remainder of 89 and 55? 89 divided by 55 is 1 leaving the remainder $89 - 55 = 34$. The next round, hence, is gcd 55 34. The remainder of 55 and 34 is $55 - 34 = 21$. We recurse once again, this time with gcd 34 21. The remainder of 34 and 21 is $34 - 21 = 13$. The next step, hence, is gcd 21 13, which leads to the remainder $21 - 13 = 8$. As you see, this gets quite boring, but we are not done yet, since the next round gcd 13 8 forces us to call the function again with 8 and $13 - 8 = 5$, which then leads to gcd 5 3, subsequently to gcd 3 2 and then to gcd 2 1. The division of 2 by 1 is 2 leaving no remainder and, finally, we call gcd 1 0, which reduces immediately to 1.

Apparently, we got in some kind of trap. The first pair of numbers, 89 and 55, leads to a sequence of numbers, where every number is the sum of its two predecessors: $1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8, 5 + 8 = 13, 8 + 13 = 21, 13 + 21 = 34, 21 + 34 = 55, 34 + 55 = 89$. We entered with 89 and 55 and computed the remainder. Since the difference of 89 and 55 is less than 55, the remainder between these two number is just the difference $89 - 55$. That way, we got to the next pair, 55 and 34, for which the same is true, *viz.* that the remainder is just the difference between the two and so we continued step for step until we finally reached $(2, 1)$.

This sequence is well known. It was used by the Italian mathematician Leonardo Pisano, better known as Fibonacci (*Filius*, that is, son of Bonaccio), as an arithmetic exercise in his *Abacus* ("calculating") book, which was published in 1202. The sequence is the solution to an exercise with the following wording: "How many pairs of rabbits can be produced from a single pair in a year's time if every fertile pair produces a new pair of offsping per month and every pair becomes fertile in the age of one month?" We start with 1 pair, which produces one offspring after one month, yielding 2 pairs; in the second month, the first pair produces one more offsping, hence we have 3 pairs. In the third month, we have 2 fertile pairs producing each 1 more pair and we, hence, have 5 pairs. This, quickly, becomes confusing. Here a table that gives an overview of what happens during the first year:

month	1	2	3	4	5	6	7	8	9	10	11	12
new pairs	1	1	2	3	5	8	13	21	34	55	89	144
total	1	2	4	7	12	20	33	54	88	143	232	376

This means that, in month 1, there is 1 new pair; in month 2, there is another new pair; in month 3, there are 2 new pairs; in month 4, there are 3 new pairs; in month 5, there are 5 new pairs; ...; in month 12, there are 144 new pairs. This is the Fibonacci sequence, whose first 12 values are given in the second row. The answer to Fibonacci's question consists in summing the sequence up: $\sum_{k=2}^{12} F_k = 375$. This can be seen in the third row of the table, which shows the total number of rabbit pairs for each month. Since this sum includes the first pair, which was already there, we must subtract one from the values in this row to come to the correct result.

The Fibonacci function can be defined as:

$$F_n = \begin{cases} 0 & \text{if n = 0} \\ 1 & \text{if n = 1} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The Fibonacci sequence is explicitly defined for 0 and 1 (since, of course, 0 and 1 do not have two predecessor from which they could be derived) and for all other numbers recursively as the sum of the Fibonacci numbers of its two predecessors. In Haskell this looks like:

```
fib :: Natural → Natural
fib 0 = 0
fib 1 = 1
fib n = fib (n − 1) + fib (n − 2)
```

Many people have studied the Fibonacci sequence, following Fibonacci, but also independently and even before it was mentioned in the *Abacus Book*. The sequence has the astonishing habit of popping up in very different contexts in the study of mathematics, nature, arts and music. Many mathematical objects, however, have this surprising – or, for some at least, annoying – property.

The first known practical application of the Fibonacci sequence in Europe appeared in an article of the French mathematician Gabriel Lamé in 1844 that identified the worst case of the Euclidian gcd algorithm as any two subsequent numbers of the Fibonacci sequence. This remarkable paper is also considered as the earliest study in computational complexity theory, a discipline that would be established only 120 years later.[1] Lamé gave the worst case of the algorithm as n for $\gcd(F_{n+2}, F_{n+1})$. Let us check if this is true for our example above. We started with 89 and 55, which correspond to F_{11} and F_{10}. According to Lamé, we would then need 9 steps to terminate the algorithm. The pairs of numbers we applied are: $(89, 55), (55, 34), (34, 21), (21, 13), (13, 8), (8, 5), (5, 3), (3, 2), (2, 1), (1, 0)$, which are 10 pairs of numbers and indeed 9 steps. But why is this so and is it always true or just for the sequence, we happen to look at?

We can answer the first question by observing the recursive nature of the Euclidian algorithm. When there is a pair of subsequent Fibonacci numbers that needs n steps, then the pair of the next subsequent Fibonacci numbers will reduce to the first pair after one round of gcd and, thus, needs $n + 1$ steps: all the steps of the first pair plus the one for itself. This is just the structure of mathematical induction, which leads us to the proof. We choose $(2, 1)$ as the base case, which is (F_3, F_2) and, as we have seen, needs one step to result in the trivial case $(1, 0)$. If the proposition that $\gcd(F_{n+2}, F_{n+1})$ needs n steps is true, then, for the case $n = 1$, F_{n+2} is F_3, which is 2, and F_{n+1} is F_2, which is 1. Therefore, the base case $(2, 1)$ fulfils the rule.

Now assume that we have a case for which it is true that the number of steps of $\gcd(F_{n+2}, F_{n+1})$ is n. Then we have to show that the number of steps of $\gcd(F_{n+3}, F_{n+2})$ is $n+1$. According to its definition, gcd for a pair of numbers (a, b) is $(b, a \bmod b)$. For subsequent Fibonacci numbers (as we have already shown informally above), $a \bmod b$ is identical to $a - b$ (except for the case where $b = 1$). After one step with $a = F_{n+3}$ and $b = F_{n+2}$, we therefore have:

$$\gcd(F_{n+2}, F_{n+3} - F_{n+2}).$$

We can substitute F_{n+3} in this formula according to the definition of the Fibonacci sequence, $F_n = F_{n-2} + F_{n-1}$, by $F_{n+1} + F_{n+2}$:

$$\gcd(F_{n+2}, F_{n+1} + F_{n+2} - F_{n+2}),$$

which, of course, simplifies to

$$\gcd(F_{n+2}, F_{n+1}).$$

This shows that we can reduce $\gcd(F_{n+3}, F_{n+2})$ to $\gcd(F_{n+2}, F_{n+1})$ in one step and that concludes the proof.

---

[1] There are of course always predecessors. The relation between gcd and the Fibonacci sequence was, according to Knuth, already discussed in a paper by a French mathematician called Léger in 1837, and an analysis of the run time behaviour of the Euclidan algorithm was already presented in a paper by another French mathematician called Reynaud in 1811.

There is much more to say about this delightful sequence, and we are even far away from the conclusions of Lamé's paper. Unfortunately, we have not yet acquired the tools to talk about these upcoming issues in a meaningful way. But, very soon, we will have. In the meanwhile, you might try to discover the Fibonacci sequence in other objects we will meet on our way, for example, in a certain very strange triangle.

4.5. Factorial

A fundamental concept in mathematics, computer science and also real life is the idea of *permutation*, variations in the order of a sequence of objects. Shuffling a card deck would for instance create permutations of the original arrangement of the cards. The possible outcomes of a sports event, the order in which the sprinters in a race arrive or the final classification of a league where all teams play all others, is another example.

For the list $[1, 2, 3]$ (in Haskell notation), the following permutations are possible: $[1, 2, 3]$ (this is the identity), $[2, 1, 3]$, $[2, 3, 1]$, $[1, 3, 2]$, $[3, 1, 2]$ and $[3, 2, 1]$.

Let us look at how to construct all permutations of a given sequence. The simplest case is the empty list that allows only one arrangement: $\mathit{permutations}\ [\,] = [[\,]]$. From this base case on, we can easily create permutations of longer lists, simply inserting new elements at every possible position within the permutations. The permutations of a list with one elmenent, for instance, would be constructed by inserting this element, say x, in all possible positions of all possible permutations of the empty list, trivially yielding: $[[x]]$. Now, when we add one more element, we get: $[[y, x], [x, y]]$, first adding the new element y in front of the existing element x and, second, adding it behind x. We now easily create the permutations of a list with three elements by simply inserting the new element z in all possible positions of these two sequences, which, for the first, gives: $[z, y, x], [y, z, x], [y, x, z]$ and for the second: $[z, x, y], [x, z, y], [x, y, z]$. Compare this pattern to the permutations of the list $[1, 2, 3]$ above with $z = 1, y = 2$ and $x = 3$.

Let us implement the process of inserting a new element at any possible position of a list in Haskell using the *cons* operator (:):

$$\begin{aligned}&\mathit{insall} :: a \to [a] \to [[a]]\\ &\mathit{insall}\ p\ [\,] \qquad\ = [[p]]\\ &\mathit{insall}\ p\ (x : \mathit{xs}) = (p : x : \mathit{xs}) : (\mathit{map}\ (x{:})\ \$\ \mathit{insall}\ p\ \mathit{xs})\end{aligned}$$

As base case, we have p, the new element, added to the empty list, which trivially results in $[[p]]$. From here on, for any list of the form $x{:}\mathit{xs}$, we add p in front of the list $(p{:}x{:}\mathit{xs})$ and then repeat the process for all possible reductions of the list until we reach the base case. In each recursion step, we add x, the head of the original list, in front of the resulting lists. Imagine this for the case $p = 1$, $x = 2$ and $\mathit{xs} = \{3\}$: We first create $[1, 2, 3]$ by means of $p : x : \mathit{xs}$; we then enter *insall* again with $p = 1$, $x = 3$ and $\mathit{xs} = \{\}$, which creates $1 : 3 : [\,]$, to which later, when we return, 2, the x of the previous step, is inserted, yielding $[2, 1, 3]$. With the next step, we hit our base case *insall* 1 [ ] = [[1]]. Returning to the step with $x = 3$, mapping $(x{:})$ gives $[3, 1]$ and, one step further back, $[2, 3, 1]$. We, hence, have created three cases: $[[1, 2, 3], [2, 1, 3], [2, 3, 1]]$ inserting 1 in front of the list, in the middle of the list and at the end.

To generate all possible permutations we would need to apply *insall* to *all* permutations of the input list, that is not only to $[2, 3]$ as above, but also to $[3, 2]$. This is done by the following function:

```
perms :: [a] → [[a]]
perms [] = [[]]
perms (x : xs) = concatMap (insall x) $ perms xs
```

Called with $[1, 2, 3]$, the function would map *insall* 1 on the result of *perms* 2 : [3]. This, in its turn, would map *insall* 2 onto *perms* 3 : [ ]. Finally, we get to the base case resulting in [[ ]]. Going back the call tree, *insall* 3 would now be called on the empty set yielding [3]; one step further back, *insall* 2 would now result in $[[2, 3], [3, 2]]$. Mapping *insall* 1 finally on these two lists leads to the desired result.

You will have noticed that we are using the function *concatMap*. The reason is that each call of *insall* creates a list of lists (a subset of the possible permutations). Mapping *insall* 1 on the permutations of $[2, 3]$, for instance, creates two lists, one for each permutation ($[2, 3]$ and $[3, 2]$): $[[1, 2, 3], [2, 1, 3], [2, 3, 1]]$ and $[[1, 3, 2], [3, 1, 2], [3, 2, 1]]$. We could use the function *concat* to merge the lists together, like: *concat*\$*map* (*insall x*)\$ *perms xs*; *concatMap* is much more convenient: it performs mapping and merging in one step.

We have not yet noted explicitly that, when talking about permutations, we treat sequences as Haskell lists. Important is that the elements in permutation lists are distinct. In a list like $[1, 2, 2]$, we cannot distinguish the last two elements leading to errors in counting possible permutations. In fact, when we say *sequence*, we mean an ordering of the elements of a *set*. Sets, by definition, do not contain duplicates. We will look at sets more closely in the next section.

So, how many possible permutations are there for a list with n elements? We have seen that for the empty list and for any list with only one element, there is just one possible arrangement. For a list with two elements, there are two permuntations ($[a, b], [b, a]$). For a list with three elments, there are six permutations. Indeed, for a list with three elements, we can select three different elements as the head of the list and we then have two possible permutations for the tail of each of these three list. This suggests that the number of permutations is again a recursive sequence of numbers: for a list with 2 elements, there are 2×1 possible permutations; for a list with 3 elements, there are 3×2 possible permutations or, more generally, for a list with n elements, there are n times the number of possiblilities for a list with $n - 1$ elements. This function is called *factorial* and is defined as:

$$n! = \prod_{k=1}^{n} k. \tag{4.14}$$

We can define factorial in Haskell as follows:

```
fac :: (Num a, Eq a) ⇒ a → a
fac 0 = 1
fac n = n ∗ fac (n − 1)
```

There is sometimes confusion about the fact that 0! is 1 and not, as one might expect, 0. There are however good arguments for this choice. The first is that the empty list is something that we can present as an input to a function creating permutations. If the output were nothing, then the empty list would have vanished by some mysterious trick. The output should therefore be the empty list again and, thus, there is exactly one possible permutation for the empty list.

Another argument is that, if 0! were 0, we could not include 0 into the recursive definition of factorial. Otherwise, the result of any factorial would be zero! The inversion of factorial, *i.e.*

$$n! = \frac{(n+1)!}{n+1}, \tag{4.15}$$

would not work either. 4! is for instance $\frac{5!=120}{5} = 24$, 3! is $\frac{4!=24}{4} = 6$, $2! = \frac{3!=6}{3} = 2$, $1! = \frac{2!=2}{2} = 1$ and, finally, $0! = \frac{1!=1}{1} = 1$.

The first factorials, which you can create by *map fac* $[0\,..\,7]$, are: $1, 1, 2, 6, 24, 120, 720, 5040$. They, then, increase very quickly, 10!, for instance, is 3 628 800. Knuth mentions that this value is a rule of thumb for the limit of what is reasonably computable. Algorithms that need more than 10! steps, quickly get out of hand, consuming too much space or time. Techniques to increase the available computational power may push this limit a bit ahead, but factorial grows even faster than Moore's law, drawing a definite line for computability.

Unfortunately, no closed form of the factorial function is known. There are approximations, at which we will look later in this book, but to obtain the precise value, a recursive computation is necessary, making factorial an expensive operation.

But let us have another look at permutations, which are very interesting beast. In the literature, different notations are used to describe permutations. A very simple, but quite verbose one is the *two-line* notation used by the great French mathematician Augustin-Louis Cauchy (1789 – 1857). In this notation, the original sequence is given in one line and the resulting sequence in a second line, hence, for a permutation σ:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 3 & 1 \end{pmatrix}.$$

According to this definition, the permutation σ would substitute 2 for 1, 5 for 2, 4 for 3 and 3 for 4 and, finally, 1 for 5. The alternative *tuple notation* would just give the second line as (2,5,4,3,1) and assume a *natural* ordering for the original sequence. This notation is useful, when several permutations on the same sequence are discussed. The original sequence would be introduced once, and afterwards only the variations are given.

More elegant, however, is the *cycle notation*, which describes the effect of subsequent applications of σ. In the example above, you see, for instance, that one application of σ on 1 would yield 2, *i.e.* 2 takes the place of 1. Another application of σ, *i.e.* the application on 2 in the second line, would result in 5 (since $\sigma(2) = 5$). The next application, this time on 5, would put 1 back into place (since $\sigma(5) = 1$). These subsequent applications describe an *orbit* of the permutation σ. Each orbit is presented as a sequence of numbers in parentheses of the form $(x\ \sigma(x)\ \sigma(\sigma(x))\ \sigma(\sigma(\sigma(x)))\ \dots)$, leaving out the final step where the cycle returns to the original configuration. An element that is fixed under this permutation, *i.e.* that remains at its place, may be presented as an orbit with a single element or left out completely. The permutation σ above in cycle notation is (1 2 5)(3 4). The first orbit describes the following relations: $\sigma(1) = 2$, $\sigma(2) = 5$ and $\sigma(5) = 1$, restoring 1 in its original place. The second orbit describes the simpler relation $\sigma(3) = 4$ and $\sigma(4) = 3$. This describes the permutation σ completely.

Can we devise a Haskell function that performs a permutation given in cycle notation? We first need a function that creates a result list by replacing elements in the original list. Since orbits define substitutions according to the original list, we need to refer to this list, whenever we make a substitution in the result list. Using the result list as a reference, we would, as in the case of 2, substitute a substitution, *e.g.* 2 for 5 at the first place instead of the second place. Here is the *replace* function:

```
replace :: (Eq a) ⇒ a → a → [a] → [a] → [a]
replace _ _ [] _ = []
replace p s (y : ys) (z : zs) | y ≡ p     = s : zs
                              | otherwise = z : replace p s ys zs
```

In this function, p is the element from the original list that will be substituted, the substitute is s. We pass through the original list and the result list in parallel assuming that the result list is initially equal to the original list. When p is found in the original, s is placed at its postition and the function terminates. (Since Haskell lists, in this case, represent sequences that do not contain duplicates, we just terminate after the first substitution.) Otherwise, the value already there at this position in the resulting list is preserved and the search continues.

We will use *replace* in the definition of a function creating permutations according to a definition in cycle notation. Cycle notation is translated to Haskell as a list of lists, each inner list representing one orbit:

```
type Perm a = [[a]]
```

The *permute* function takes such a *Perm* and a list on which to perform the permutation. An orbit consisting of the empty list or of only one element is the identity and, hence, ignored. Otherwise, one orbit after the other is processed:

```
permute :: (Eq a) ⇒ Perm a → [a] → [a]
permute []         xs = xs
permute ([] : ps)  xs = permute ps xs
permute ([p] : ps) xs = permute ps xs
permute (p : ps)   xs = permute ps $ orbit (head p) xs p
  where orbit _ rs []               = rs
        orbit x rs [u]              = replace u x xs rs
        orbit x rs (p1 : p2 : pp) = orbit x (replace p1 p2 xs rs) (p2 : pp)
```

For every orbit (that contains more than one element), *permute* is applied to the result of the function *orbit*, which takes the first element of the current orbit, the input list and the current orbit as a whole. The function processes the orbit by replacing the first element by the second, the second by the third and so on. The last element is replaced by the head of the orbit, which, for this purpose, is explicitly passed to the function.

Note that each call to *orbit* and, hence, each recursion step of permute creates a result list, which is then used for the next recursion step. Since orbits do not share elements, no change in the result list made according to one orbit will be touched when processing another orbit; only elements not yet handled by the previous orbits will be changed. It is therefore safe to substitute the input list by the list resulting from processing the previous orbits.

The cyclic notation introduces the idea of composing permutations, *i.e.* applying a permutation on the result of another. The permutation above applied to itself, for instance, would yield $[5,1,3,4,2]$; applying it once again results in $[1,2,4,3,5]$. Six subsequent applications would return to the original list:

let *sigma* = *permute* $[[1,2,5]\ [3,4]]$
sigma $[1,2,3,4,5]$ is $[2,5,4,3,1]$.
sigma $[2,5,4,3,1]$ is $[5,1,3,4,2]$.
sigma $[5,1,3,4,2]$ is $[1,2,4,3,5]$.
sigma $[1,2,4,3,5]$ is $[2,5,3,4,1]$.
sigma $[2,5,3,4,1]$ is $[5,1,4,3,2]$.
sigma $[5,1,4,3,2]$ is $[1,2,3,4,5]$.

The result in the third line is funny: It is almost identical to the original list, but with 3 and 4 swapped. The two orbits of the permutation σ appear to move at different speed: the first orbit with three elements needs three applications to return to the original configuration; the second orbit with two elements needs only two applications. Apparently, 2 does not divide 3; the orbits are therefore out of sink until the permutation was performed $2 \times 3 = 6$ times.

One could think of systems of permutations (and people have actually done so), such that the application of the permutations within this system to each other, *i.e.* the composition of permutations (denoted: $a \cdot b$), would always yield the same set of sequences. Trivially, all possible permutations of a list form such a system. More interesting are subsets of all possible permutations. Let us simplifiy the original list above to $[1, 2, 3, 4]$, which has 4 elements and, hence, $4! = 24$ possible permutations. On this list, we define a set of permutations, namely

$$e = (1)(2)(3)(4) \tag{4.16}$$
$$a = (1\ 2)(3)(4) \tag{4.17}$$
$$b = (1)(2)(3\ 4) \tag{4.18}$$
$$c = (1\ 2)(3\ 4) \tag{4.19}$$

The first permutation e is just the identity that fixes all elements. The second permutation, a, swaps 1 and 2 and fixes 3 and 4. One application of a would yield $[2, 1, 3, 4]$ and two applications $(a \cdot a)$ would yield the original list again. The third permutation, b, fixes 1 and 2 and swaps 3 and 4. One application of b would yield $[1, 2, 4, 3]$ and two applications $(b \cdot b)$ would yield the original list again. The fourth permutation, c, swaps 1 and 2 and 3 and 4. It is the same as $a \cdot b$, thus creating $[2, 1, 4, 3]$ and $c \cdot c$ would return to the original list. We can now observe that all possible compositions of these permutations, create permuations that are already part of the system:

$$\begin{aligned} a \cdot a = b \cdot b &= e \\ b \cdot a \cdot b \cdot a &= e \\ a \cdot b = b \cdot a &= c \\ c \cdot c &= e \\ c \cdot a \cdot b &= e \end{aligned}$$

You can try every possible combination, the result is always a permutation that is already there. This property of composition of the set of permutations above bears some similarity with natural numbers together with the operations addition and multiplication: The result of an addition or multiplication with any two natural numbers is again a natural number, and the result of a composition of any two permutations in the system is again in the system.

Such systems of permutations, hence, are magmas (as defined in the previous chapter) where the carrier set is the set of permutations and the binary operation is composition. Furthermore, the permutation system fulfils associativity: $a \cdot (b \cdot c) = (a \cdot b) \cdot c = a \cdot b \cdot c$. So, it is also a semigroup. Since the identity permutation is part of the system, the system is also a monoid and, to be more specific, an abelian monoid, since commutativity, as well, is a property of the composition permutations.

Since we designed the system in a way that every permutation, applied to itself, restores the original sequence, such that $x \cdot y = y \cdot x = e$, there is also an inverse element to every element in the system: the inverse of a permutation is the composition with itself! $a \cdot a = e$. This means that we have found a group!

The set of all possible permutations of a sequence, trivially, is always a group and called the *symmetric group*: since all possible permutations are in the group, every possible composition of two permutations leads to a permutation that is in the group as well, so it is closed under composition; composition, as we have seen, is associative; since, again, all permutations are in the group, there is an identiy element (the permutation that fixes all elements) and, since all possible permutations are in the group, there is for each permutation a permutation that returns to the original configuration, the inverse element. These properties make the symmetric group a group.

But, of course, not all possible subsets of the symmetric group are groups. Subsets of the symmetric group that do not contain the identity are not groups; sets containing permutations that, composed with each other, yield a permutation that is not part of the set are not groups either.

4.6. Random Permutations

We have discussed how we can generate all permutations of a given sequence. But we have not discussed the much more frequent task of creating a *random* permutation of a given sequence.

Algorithms creating random permutations are relatively simple compared to those creating all permutations – if there were not the adjective *random*. Randomness, in fact, is quite a difficult issue in particular when we are thinking of ways to achieve real randomness. Randomness in mathematics is usually defined in terms of a sequence. According to the definition of the great Russian mathematician Andrey Kolmogorov (1903 – 1987) who actually axiomatised and thereby modernised probability theory, a sequence is random, when it cannot be created by a program that, interpreted as a string, is shorter than that sequence. For instance, we could look at any sequence of numbers such as $1, 2, 3, \dots$ A program to create such a sequence is just $genSeq\ n = n : genSeq\ (n + 1)$ and, obviously, much shorter than the resulting sequence.

When you think of it, it is indeed difficult to create a sequence without any *patterns* in it, such as regular distances between elements, periodic repetitions and so on. You may think of any of the sequences we have looked at so far: there was always a pattern that led to a way to define a program to generate that sequence and the program was always represented as a finite string of Haskell code that was much shorter than the sequence, which, usually, was infinite. For instance, the definitions of the Fibonacci sequence or of Factorials are much shorter than that sequences, which are infinite. But, even with finite sequences, we have the same principle. Look, for instance, at the sequence $5, 16, 8, 4, 2, 1$, which, on the first sight, appears completely random. However, there is a program that generates this as well as many other similar sequences, namely the *hailstone* algorithm:

```
hailstone :: Integer → [Integer]
hailstone 1 = [1]
hailstone n | even n    = n : hailstone (n ‘div‘ 2)
            | otherwise = n : hailstone (3 ∗ n + 1)
```

One may argue that this code is in fact longer than the resulting sequence. But it would be very easy to encode it in a more concise way, where, for instance, numerical codes represent the tokens of the Haskell language. Furthermore, the code implements the general case that creates the hailstone sequence for any number > 1. For $n = 11$, it is already a bit longer: $11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

The hailstone algorithm, by the way, always terminates with 1, independent of the number n we start with. This is the *Collatz conjecture*, named after the German mathematician Lothar Collatz (1910 – 1990), who posed the problem in 1937. It is unproven and it might be undecidable according to results from John Conway. But that is another story.

Kolmogorov randomness does not only apply to numerical sequences. When we have a sequence of symbols like $a, b, c, d, \dots$, there is either some regularity or it is not possible to define a program that does not contain the sequence itself and, hence, has no potential to be shorter than the sequence in the first place. The question arises: how do we generate a random sequence, if there is no program that generates it and is significantly shorter than that sequence? Would that not mean that, to generate n bits of randomness, we would need a program that is at least n bits long? Yes, that is basically the case. Any short deterministic program, however this program is implemented, will follow some rules and will eventually create a sequence that still bears traces of that regularity.

The only way to generate true randomness is to pick up numbers from outside of the current problem domain, that is we have to look around to find numbers from other contexts. But, careful: many numbers you see around you still contain regularities. For instance, all numbers generated with the current date as input bear regularity related to the date. It would not be a good idea to use such a date-related number to create, say, a session key for securely encrypted communication through an open channel.

Random number generators implemented in modern operating systems collect numbers that are created by the system while operating. A typical source of randomness is keystrokes. Every single keystroke creates some data that is stored in a pool for randomness from which other programs can later request some bits of randomness. To get access to true random data, thus, implies that the program requesting those data needs to interact with the operating system. Therefore, whenever we need randomness in Haskell, we need the *IO Monad*. This adds some complexity to our code; but, in fact, this complexity just reflects reality: randomness *is* complex.

In Haskell, there is a module called *System.Random* that provides functions to create random numbers, both *pseudo-random* numbers, which create sequences that appear random on the first sight, but are generated by deterministic algorithms, and true random numbers. Interesting for us in this module is the function *randomRIO*, which creates random objects within a range defined as a tuple. The call *randomRIO* $(0, 9)$, for instance, would create a random number between 0 to 9 (both included). Since *randomRIO* does not know our number type *Natural*, we would have to define a way for *randomRIO* to create random *Natural*s. It is much simpler, however, to use a type known to *randomRIO* and to convert the result afterwards. Here is a simple implementation of a function *randomNatural* that generates a random natural number:

```
randomNatural :: (Natural, Natural) → IO Natural
randomNatural (l, u) = let il = fromIntegral l
                           iu = fromIntegral u
                       in  fromIntegral < $ >
                           (randomRIO (il, iu) :: (IO Integer))
```

The range we want the result to lie in is defined by the tuple (l, u), for *lower* and *upper*. We convert the elements of the tuples to *li* and *iu*, which, as we see in an instance, are of type *Integer*. We then call the random number generator with the type signature *IO Integer* defining the output type. This output is finally converted back to natural using *fromIntegral*.

The canonical algorithm for generating random numbers is called *Fisher-Yates shuffle*, after its inventors Ronald Fisher (1890 – 1962) and Frank Yates (1902 –1994), but is also called *Knuth shuffle*, because it became popular through Knuth's masterpiece. The algorithm goes through the sequence we want to permute and, for each index i, that is the place of the element in the sequence starting from 0, it generates a random number j between 0 and $n - 1$, where n is the number of elements in the sequence. If this number is different from the current index, it swaps the elements at positions i and j.

Until now, we have worked only with lists. Lists are extremely efficient, when passing through from the head to the last. Now, however, we need to refer to other places in the list that may be ahead to the end of the sequence or behind closer to its head, depending on the value of j. Also, we have to change the list by going through it. This is essential, because, we might change the same place more than once. For the *fold*-kind of processing that was so typical for the functions we have studied so far, this would be extremely ineffecient. We therefore use another data type, a mutable vector, defined in *Data.Vector.Mutable*. First, we will look at a function that creates a mutable vector from a list:

```
createVector :: [a] → IO (V.IOVector a)
createVector xs = do v ← V.new (length xs)
                     initV v 0 xs
                     return v
  where initV _ _ []       = return ()
        initV v i (z : zs) = V.unsafeWrite v i z
                             ≫ initV v (i + 1) zs
```

We first create a new vector of the size of the list. Then we initialise this vector just passing through the list in a *map* fashion, but incrementing the index *i* at each step. We use the vector function *unsafeWrite*, which takes a vector, *v*, an index, *i*, and the value to write, *z*. The function is called *unsafe* because it does not perform a boundary check (and is, as such, much faster than its safe cousin). Since we are careful to move within the boundaries, there is no huge risk involved in using the *unsafe* version of this operation. Finally, we just return the initialised vector.

The next function does the opposite: it converts a vector back to a list:

```
vector2list :: V.IOVector a → Int → IO [a]
vector2list v n = go 0
  where go i | i ≡ n     = return []
             | otherwise = do x ← V.unsafeRead v i
                              (x:) < $ > go (i + 1)
```

The function is quite simple. It goes through the vector reading one position after the other and, when it reaches *n*, just returns the empty list. On each step, the value at position *i* is read and inserted as the head of the list that results from recursing on *go*. Now we are ready to actually implement the *kshuffle*:

```
kshuffle :: [a] → IO [a]
kshuffle xs = do let n = length xs
                 vs ← createVector xs
                 is ← randomidx n 0
                 go 0 is vs
                 vector2list vs n
  where randomidx n k | k ≡ n     = return []
                      | otherwise = do i ← randomRIO (0, n − 1)
                                       (i:) < $ > randomidx n (k + 1)
        go _ [] _        = return ()
        go k (i : is) vs = when (k ≢ i) (V.unsafeSwap vs k i)
                           ≫ go (k + 1) is vs
```

We start by creating the vector using the function *createVector* defined above. Note that, since we need it more than once, we initially store the size of the list in the variable n. Since we compute it again in *createVector*, there is potential for improvement.

In the next step, we create a list of n numbers using *randomidx*. *randomidx* calls *randomRIO* n times making each result head of the list that is constructed by recursion. Note that we do not use *randomNatural*. We will see in *go* that the results of *randomidx* are used as vector indices and, since vector indices are of type *Int*, we spare some forth and back conversions. *go* expects three arguments: an *Int* called k, a list of *Int*, these are the random indices just created, and the vector on which we are operating. For each index in the list, we swap the value at position k in the vector, which is the index in the natural ordering starting from 0, with the value at position i and continue with the recursion on *go* with $k + 1$ and the tail of the list of random indices. Finally, we call *vector2list* on the manipulated vector yielding a permutation of the input list.

One may be tempted to say that the permutation is generated by a permutation of the indices of the initial list. But do not be fooled! The random indices we are generating do not consitute, at least not necessarily, a valid permutation of the natural ordering of the input list. Each index is generated randomly – completely *independent* of the other indices. In consequence, some of the values we get back from *randomRIO*, in fact, at least theoretically, all of them, may be equal – and this is the whole point of this shuffle.

Consider the input list a, b, c, d, e with the natural ordering of positions $0, 1, 2, 3, 4$, *i.e.* at postion 0, we have a, at position 1, we have b, at position 2, we have c and so on. *randomidx* could result in a list of random indices like, for example, $2, 0, 1, 3, 4$, which would be a permutation of the natural order. However, it may also result in a list like $2, 0, 1, 1, 4$, which is not a permutation. The *kshuffle* algorithm does not require the constraint that the indices we create form a permutation of the initial order. It guarantees that the overall result is actually a permutation of the input list without such a constraint. This saves us from the trouble of checking the result of the random number generator and calling it again each time, there is a collision.

Imagine *randomidx* would create the list $1, 1, 1, 1, 1$, which we could obtain with a probability of $\frac{1}{5} \times \frac{1}{5} \times \frac{1}{5} \times \frac{1}{5} \times \frac{1}{5} = \frac{1}{5^5} = \frac{1}{3125}$. We now *go* through the natural positions k, $0 \dots 4$ and the vector initially representing the list a, b, c, d, e. It is essential to realise that operations on a mutable vector are *destructive*, that is all operations are performed on the current state of the vector, which changes from step to step, such that the output of each step is the input to the next step. What happens is the following:

1. We swap position 0 and 1 resulting in b, a, c, d, e;
2. We do not do anything, because the indices k and i are both 1 in the second step, maintaining b, a, c, d, e;
3. We swap positions 2 and 1 resulting in b, c, a, d, e;
4. We swap positions 3 and 1 resulting in b, d, a, c, e;

5. We swap positions 4 and 1 resulting in b, e, a, c, d,

resulting overall in a valid permutation of the input list.

4.7. Binomial Coefficients

Closely related to permutations are problems of selecting a number of items from a given set. Whereas permutation problems have the structure of shuffling cards, selection problems have that of dealing cards. This analogy leads to an intuitive and simple algorithm to find all possible selections of k out of a set with n elements by taking the first k objects from all possible permutations of this set and, afterwards, removing the duplicates. Consider the set $\{1, 2, 3\}$ and let us find all possible selections of two elements of this set. We start by choosing the first two elements of the given sequence and get $\{\{1, 2\}\}$. Now we create a permutation: $\{2, 1, 3\}$ and, again, take the first two elements. The result set is now: $\{\{1, 2\}, \{2, 1\}\}$. We continue with the next permuation $\{2, 3, 1\}$, which leads us to the result set $\{\{1, 2\}, \{2, 1\}, \{2, 3\}\}$. Going on this way – and we already have defined an algorithm to create all possible permutations of a set in the previous section – we finally get to the result set $\{\{1, 2\}, \{2, 1\}, \{2, 3\}, \{3, 2\}, \{3, 1\}, \{1, 3\}\}$. Since, as we know from the previous section, there are $3! = 6$ permutations, there are also six sequences with the first k elements of these six permutations. But, since we want unique selections, not permutations of the same elements, we now remove the duplicates from this result set and arrive at $\{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$, that is three different selections of two elements out of three.

This algorithm suggests that the number of k selections out of n elements is somehow related to the factorial function. But, obviously, the factorial is too big a result, we have to reduce the factorial by the number of the permutations of the results. Let us think along the lines of permutation: we have 3 ways to select 1 object out of 3: $\{\{1\}, \{2\}, \{3\}\}$. For the factorial, we said that we now combine all permutations of the remaining 2 objects with this 3 possible solutions and compute the number of these permutations as 3×2. However, since order does not matter, the first selection conditions the following selections. After the first step, we seemingly have two options for each of the first selections in step 2:

step 1	step 2
1	$\{2, 3\}$
2	$\{1, 3\}$
3	$\{1, 2\}$

But note that, when we select 2 in the first row, the option 1 in the second row will vanish, since we already selected $\{1, 2\}$, which is the same as $\{2, 1\}$. Likewise, when we select 3 in the first row, we cannot select 1 in the third row because, again, $\{1, 3\}$ is the same as $\{3,1\}$. It, therefore, would be much more appropriate to represent our options as in the following table:

step 1	step 2
1	$\{2,3\}$
2	$\{3\}$
3	$\{\}$

At the beginning, we are completely free to choose any element, but when we come to the second, the options are suddenly reduced and at the third step there are no options left at all. For the case 2 out of 3, we see that the first selection halves our options in the second step. This suggests that we have to divide the number of options per step. With permutation, we had $n \times (n-1)$, but with selection, we apparently have something like $n \times \frac{n-1}{2}$, which, for the case 2 out of 3, is $3 \times \frac{3-1}{2} = 3 \times \frac{2}{2} = 3 \times 1 = 3$. When we continue this scheme, considering that each choice that was already made conditions the next choice, we get a product of the form: $\frac{n}{1} \times \frac{n-1}{2} \times \frac{n-2}{3} \times \dots$ Selecting 3 out of 5, for instance, is: $\frac{5}{1} \times \frac{4}{2} \times \frac{3}{3} = 10$. This leads to the generalised product for k out n: $\frac{n}{1} \times \frac{n-1}{2} \times \dots \times \frac{n-(k-1)}{k}$. This product is known as the binomial coefficient $\binom{n}{k}$ pronounced *n choose k*.

We easily see that the part below the fraction line is $k!$ The part above the line is a partial factorial of n, called falling factorial or *to-the-k^{th}-falling*:

$$n^{\underline{k}} = n \times (n-1) \times \dots \times (n-k+1) = \prod_{j=1}^{k} n+1-j. \tag{4.20}$$

We, therefore, can represent the binomial coefficient as either:

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j} \tag{4.21}$$

or:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{4.22}$$

But there is still another formula, which, even though less efficient in terms of computational complexity, is often used to ease proofs involving binomial coefficients and which is closer to our first intuition that the selection is somehow related to factorials reduced by some value:

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!}. \tag{4.23}$$

It can be seen immediately that this formula is equivalent to equation 4.22, whenever $k \leq n$, since the values of $n!$ in the numerator cancel out with the values of $(n-k)!$ in the denominator. Indeed, $n!$ could be split into two halves (which are not necessarily equal of course), the upper product $n^{\underline{k}}$ $(n \times (n-1) \times \dots \times (n-k+1))$ and the lower product $(1 \times 2 \times \dots \times (n-k))$. By cancelling out the lower half, we remove the lower product from numerator and denominator and are left with the falling factorial in the numerator.

We could have derived equation 4.22 much more easily with a different kind of reasoning: Given a set with n elements, there are $n^{\underline{k}}$ permutations of k elements of this set. There are n ways to choose the first element, $n-1$ ways to choose the second element and so on and $n-k+1$ ways to choose the k^{th} element. Obviously, we could reach the same result, all permutations of k elements out of n, by first selecting these k elements and then create all possible permutations of these k elements. The number of possibilities of choosing k out of n is the binomial coefficient, $\binom{n}{k}$, which we would like to derive. The possible permutations of these k elements is of course $k!$ We now have to combine these two steps: We have for any selection of k elements out of n $k!$ permutations, that is $\binom{n}{k} \times k!$ Since this processing has the same result as choosing all permutations of k out of n in the first place, we come up with the equation:

$$n^{\underline{k}} = \binom{n}{k} \times k! \tag{4.24}$$

To know what the expression $\binom{n}{k}$ is we just divide $k!$ on both sides of the equation and get equation 4.22:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{4.25}$$

Let us look at some concrete values of the binomial coefficients: $\binom{n}{0} = \binom{n}{n} = 1$ and for $k < 0$ or $k > n$: $\binom{n}{k} = 0$. For $0 \leq k \leq n$, for instance: $\binom{3}{2} = 3$, $\binom{4}{2} = 6$, $\binom{4}{3} = 4$, $\binom{5}{2} = 10$, $\binom{5}{3} = 10$. We can arrange the results in a structure, called Pascal's Triangle, after the great French mathematician and philosopher Blaise Pascal (1623 – 1662) who used binomial coefficients to investigate probabilities and, in the process, created a new branch of mathematics, namely probability theory:

0										1									
1									1		1								
2								1		2		1							
3							1		3		3		1						
4						1		4		6		4		1					
5					1		5		10		10		5		1				
6				1		6		15		20		15		6		1			
7			1		7		21		35		35		21		7		1		
8		1		8		28		56		70		56		28		8		1	
9	1		9		36		84		126		126		84		36		9		1

In this triangle, each row represents the coefficients for one specific value of n in $\binom{n}{k}$. The left-most value in each line represents the value $\binom{n}{0} = 1$ and the right-most value is $\binom{n}{n} = 1$. The values between the outermost ones represent the values for $\binom{n}{1} \dots \binom{n}{n-1}$. The line for $n = 2$, *i.e.* the third line, for instance, shows the values $\binom{2}{0} = 1$, $\binom{2}{1} = 2$ and $\binom{2}{2} = 1$. The line for $n = 3$ shows the values $\binom{3}{0} = 1$, $\binom{3}{1} = 3$, $\binom{3}{2} = 3$ and $\binom{3}{3} = 1$, the line for $n = 4$ shows the values $\binom{4}{0} = 1$, $\binom{4}{1} = 4$, $\binom{4}{2} = 6$, $\binom{4}{3} = 4$ and $\binom{4}{4} = 1$ and so on.

This extraordinary triangle reveals many "hidden" relations of the binomial coefficients. We can observe, to start with this one, that the triangle is horizontally symmetric, *i.e.* $\binom{3}{1} = \binom{3}{2} = 3$, $\binom{6}{2} = \binom{6}{4} = 15$, $\binom{7}{2} = \binom{7}{5} = 21$ or, in general, $\binom{n}{k} = \binom{n}{n-k}$. This is a strong hint how we can optimise the computation of the binomal coefficients. Indeed, whenever k in $\binom{n}{k}$ is more than the half of n, we can use the corresponding value from the first half of k's, *i.e.*

$$\binom{n}{k} = \begin{cases} \binom{n}{n-k} & \text{if } 2k > n \\ \prod_{j=0}^{k} \frac{n+1-j}{j} & \text{otherwise} \end{cases} \tag{4.26}$$

Thank you, Triangle!

Another observation is that every coefficient is the sum of two preceding coefficients, namely the one left-hand up and the one right-hand up, *e.g.* $\binom{3}{1} = \binom{2}{0} + \binom{2}{1} = 3$, $\binom{4}{2} = \binom{3}{1} + \binom{3}{2} = 6$, $\binom{5}{2} = \binom{4}{1} + \binom{4}{2} = 10$ or, in general:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}. \tag{4.27}$$

This identity called *Pascal's Rule* does not only help us to guess the next value in a sequence, but is also the basis for techniques to manipulate equations involving binomial coefficients.

A real light bulb moment, however, comes when realising the relation of binomial coefficients to multiplication. We discussed several times already that there are certain patterns in multiplication, which now turn out of have a name: *binomial coefficient.* Indeed, this relation is one of the most important theorems in mathematics, the *binomial theorem*, which we will formulate in a second. First, let us look at the multiplication pattern. The distributive law tells us that

$$(a+b)(c+d) = ac + ad + bc + bd. \tag{4.28}$$

Now, what happens if $a = c$ and $b = d$? We would then get:

$$(a+b)(a+b) = aa + ab + ba + bb, \tag{4.29}$$

which is the same as

$$\mathbf{(a+b)(a+b) = a^2 + 2ab + b^2}. \tag{4.30}$$

When we now multiply $(a+b)$ with this result, we get:

$$\begin{aligned} (a+b)(a^2+2ab+b^2) = a^3 + 2a^2b + ab^2 + ba^2 + 2ab^2 + b^3 &= \\ a^3 + 2a^2b + ba^2 + ab^2 + 2ab^2 + b^3 &= \\ \mathbf{a^3 + 3a^2b + 3ab^2 + b^3} & \end{aligned} \tag{4.31}$$

Multiplied with $(a+b)$ once again:

$$\begin{aligned} (a+b)(a^3 + 3a^2b + 3ab^2 + b^3) &= \\ a^4 + 3a^3b + 3a^2b^2 + ab^3 + ba^3 + 3a^2b^2 + 3ab^3 + b^4 &= \\ a^4 + 3a^3b + ba^3 + 3a^2b^2 + 3a^2b^2 + ab^3 + 3ab^3 + b^4 &= \\ \mathbf{a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4} & \end{aligned} \tag{4.32}$$

The coefficients in these formulas, as you can see, equal the binomial coefficients in Pascal's Triangle. The Triangle can thus be interpreted as results of power functions:

$$\begin{gathered} (a+b)^0 = 1 \\ (a+b)^1 = 1a + 1b \\ (a+b)^2 = 1a^2 + 2ab + 1b^2 \\ (a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3 \\ (a+b)^4 = 1a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + 1b^4 \end{gathered}$$

$$(a+b)^5 = 1a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + 1b^5$$

$$\dots$$

This, in general, is the binomial theorem:

$$\begin{aligned}(x+y)^n &= \binom{n}{0}x^ny^0 + \binom{n}{1}x^{n-1}y^1 + \dots + \binom{n}{n}x^0y^n \\ &= \sum_{k=0}^{n}\binom{n}{k}x^ky^{n-k}\end{aligned} \tag{4.33}$$

But why is this so? According to multiplication rules, the multiplication of two factors $(a+b)(c+d)$ yields a combination of each of the terms of one of the factors with the terms of the other factor: $ac+ad+bc+bd$. If $a=c$ and $b=d$, we will create combinations of terms with themselves: $aa+ab+ba+bb$. How many ways are there to combine a with a in $(a+b)(a+b)$? There is exactly one way, because the a of the first factor will find exactly one a in the second factor. But how many ways are there to combine a and b? Well, the a in the first factor will find one b in the second, and the b in the first factor will find one a in the second. There are hence two ways to combine a and b and we could interpret these two combinations as two different *strings*, the string ab and the string ba. We know that there are $\binom{2}{1}=2$ different ways to select one of these strings: either ab or ba. Since these strings represent products of a and b and, according to the commutative law, the order of the factors does not matter, we can just add them up, which leaves us with a coefficient that states exactly how many strings of homogeneous as and bs there are in the sum.

There is a nice illustration of this argument: Let us look at the set of the two numbers $\{1,2\}$. There are two possibilities to select one of these numbers: 1 or 2. Now, we could interpret these numbers as answer to the question “What are the positions where one of the characters ’a’ and ’b’ can be placed in a two-character string?” The answer is: either at the beginning or at the end, *i.e.* either $\mathbf{a}b$ or $b\mathbf{a}$. For $(a+b)^3$, this is even more obvious. Compare the positions of the a’s in terms with two a’s with the possible selections $\{\{1,2\},\{1,3\},\{2,3\}\}$ of two out of the set $\{1,2,3\}$: $(a+b)(aa+ab+ba+bb) = aaa + \mathbf{aa}b + \mathbf{a}b\mathbf{a} + abb + b\mathbf{aa} + bab + bba + bb$.

This is a subtle argument. To assure ourselve that the theorem really holds for all n, we should try a proof by induction. We have already demonstrated that it indeed holds for several cases, like $(a+b)^0$, $(a+b)^1$, $(a+b)^2$ and so on. Any of these cases serves as base case. Assuming the base case holds, we will show that

$$(a+b)^{n+1} = \sum_{k=0}^{n+1}\binom{n+1}{k}a^kb^{n+1-k}. \tag{4.34}$$

We start with the simple equation

$$(a+b)^{n+1} = (a+b)^n(a+b) \tag{4.35}$$

and then reformulate it replacing $(a+b)^n$ by the base case:

$$(a+b)^{n+1} = \left(\sum_{k=0}^{n}\binom{n}{k}a^kb^{n-k}\right)(a+b). \tag{4.36}$$

We know that, to multiply a sum with another sum, we have to distribute all the terms of one sum over all terms of the second sum. This is, we multiply a with the summation and then we multiply b with the summation. In the first case, the exponents of a within the summation are incremented by one, in the second case, the exponents of b are incremented by one:

$$(a+b)^{n+1} = \sum_{k=0}^{n}\binom{n}{k}a^{k+1}b^{n-k} + \sum_{k=0}^{n}\binom{n}{k}a^kb^{n+1-k}. \tag{4.37}$$

The second term looks already quite similar to the case in equation 4.34, both have a^kb^{n+1-k}. Now, to make the first term match as well, we will use one of those *tricks* that make many feel that math is just about pushing meaningless symbols back and forth. Indeed, since we are working with sums here, the proof involves much more technique than the proofs we have seen so far. The purpose, however, is still the same: we want to show that we can transform one formula into another by manipulating these formulas according to simple grammar rules. That this has a very technical, even *tricky* flavour is much more related to the limitations of our mind that does not see through things as simple as numbers, but has to create formal apparatus not to get lost in the dark woods of reasoning.

Well, what is that trick then? The trick consists in raising the k in the summation index and to change the terms in the summation formula accordingly, that is, instead of a^{k+1}, we want to have a^k and we achieve this, by not letting k run from 0 to n, but from 1 to $n+1$:

$$(a+b)^{n+1} = \sum_{k=1}^{n+1}\binom{n}{k-1}a^kb^{n+1-k} + \sum_{k=0}^{n}\binom{n}{k}a^kb^{n+1-k}. \tag{4.38}$$

Please confirm for yourself with pencil and paper that the first summation in equations 4.37 and 4.38 is the same:

$$\sum_{k=0}^{n}\binom{n}{k}a^{k+1}b^{n-k}=\sum_{k=1}^{n+1}\binom{n}{k-1}a^{k}b^{n+1-k}$$

All we have done is pushing the index of the summation one up and, to maintain the value of the whole, reducing k by one in the summation formula.

Now we want to combine the two sums, but, unfortunately, after having pushed up the summation index, the two sums do not match anymore. Apparently, while trying to solve one problem, we have created another one. But hold on! Let us try a bit and just take the case $k = n + 1$ in the first term and the case $k = 0$ in the second term out. The case $k = n + 1$ corresponds to the expression $\binom{n}{n+1-1}a^{n+1}b^{n+1-(n+1)}$, which, of course, is simply a^{n+1}, since $\binom{n}{n+1-1} = \binom{n}{n} = 1$ and $b^{n+1-(n+1)} = b^{n+1-n-1} = b^0 = 1$. Accordingly, the case $k = 0$ in the second term corresponds to $\binom{n}{0}a^0b^{n+1-0} = b^{n+1}$. When we combine all those again, we get to:

$$(a+b)^{n+1}=a^{n+1}+\sum_{k=1}^{n}\binom{n}{k-1}a^{k}b^{n+1-k}+\sum_{k=1}^{n}\binom{n}{k}a^{k}b^{n+1-k}+b^{n+1}. \tag{4.39}$$

The next step provides you with a test of how well you have internalised the distributive law. The sum of the two summations has the form: $(\alpha c + \alpha d) + (\beta c + \beta d)$, where $\alpha = \binom{n}{k-1}$ and $\beta = \binom{n}{k}$ and c and d represent different steps of the summations, *i.e.* c is a^kb^{n+1-k} for $k = 1$ and d the same for $k = 2$ and so on. Please make sure that you see this analogy!

By applying the distributive law once, we get to $\alpha(c + d) + \beta(c + d)$. This is really fundamental – please make sure you get to the same result by distributing α and β over their respective $(c + d)$!

Now we apply the distributive law once again taking $(c + d)$ out: $(\alpha + \beta)(c + d)$. Please make sure again that this holds for you by distributing $(c + d)$ over $(\alpha + \beta)$!

When we substitute α and β by the binomial coefficients, we get $(\binom{n}{k-1} + \binom{n}{k})(c + d)$, right? In the next equation, we have just applied these little steps:

$$(a+b)^{n+1}=a^{n+1}+\sum_{k=1}^{n}\left(\binom{n}{k-1}+\binom{n}{k}\right)a^{k}b^{n+1-k}+b^{n+1}. \tag{4.40}$$

Now you might recognise Pascal's rule given in equation 4.27 above. Indeed, the Almighty Triangle tells us that $\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$. In other words, we can simplify the equation to

$$(a+b)^{n+1} = a^{n+1} + \sum_{k=1}^{n} \binom{n+1}{k} a^k b^{n+1-k} + b^{n+1}. \tag{4.41}$$

Finally, we integrate the special cases $k = 0$ and $k = n + 1$ again, just by manipulating the summation index:

$$(a+b)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} a^k b^{n+1-k} \tag{4.42}$$

and we are done, since – using some mathematical trickery – we have just derived equation 4.34.

Coming back to the question of how to implement binomial coefficients efficiently, we should compare the two alternatives we have already identified as possible candidates, *viz.* equations 4.21 and 4.22, which are repeated here for convenience:

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j}, \tag{4.43}$$

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{4.44}$$

The first option performs k divisions and $k - 1$ multiplications: one division per step and the multiplications of the partial results, that is $k + k - 1 = 2k - 1$ operations in total, not counting the sum $n + 1 - j$, which is a minor cost factor.

The second option performs $k - 1$ multiplications for $n^{\underline{k}}$, $k - 1$ multiplications for $k!$ and one division, hence, $k - 1 + k - 1 + 1 = 2k - 1$ operations. That looks like a draw.

In general terms, there is an argument concerning implementation strategy in favour of the first option. With the second option, we first create two potentially huge values that must be kept in memory, namely $n^{\underline{k}}$ and $k!$. When we have created these values, we reduce them again dividing one by the other. The first option, in contrast, builds the final result by stepwise incrementation without the need to create values greater than the final result. So, let us implement the first option:

```
choose :: Natural → Natural → Natural
choose n 0 = 1
choose n 1 = n
choose n k | k > n       = 0
           | 2 ∗ k > n   = choose n (n − k)
           | otherwise   = go 1 1
```

$$\begin{array}{lll} \textbf{where}\ go\ m\ i \mid i > k & = m \\ \qquad\qquad\quad \mid otherwise & = go\ (m * (n - k + i)\ \text{`}div\text{`}\ i)\ (i + 1) \end{array}$$

The implementation is straight forward. The function *choose* is defined over natural numbers, so we do not have to deal with negative numbers. The first parameter corresponds to n, the second one to k in $\binom{n}{k}$. Whenever $k = 0$, the result is 1 and if $k = 1$, the result is n. For all other cases, we first test if $k > n$; if so, the result is just 0. Otherwise, we make the distinction $2k > n$ and if so, we calculate *choose* for n and $n - k$, *e.g.* $\binom{5}{4} = \binom{5}{1}$. Otherwise we build the product using the function *go* that is defined as follows: If $i > k$, we use m, otherwise we recurse with the result of $\frac{m \times (n-k+i)}{i}$ and $i + 1$.

Let us look at the example $\binom{5}{3}$. We start with *go* 1 1, which expands to

$go\ (1 * (5 - 3 + 1)\ \text{`}div\text{`}\ 1)\ (1 + 1)$,

which is *go* 3 2. This, in its turn, expands to

$go\ (3 * (5 - 3 + 2)\ \text{`}div\text{`}\ 2)\ (2 + 1)$,

which equals *go* 6 3, expands to

$go\ (6 * (5 - 3 + 3)\ \text{`}div\text{`}\ 3)\ (3 + 1)$

and results in *go* 10 4. Since i is now greater than k, $4 > 3$, we just get back m, which is 10 and, thus, the correct result.

A word of caution might be in place here. The *choose* function above does not implement the product in the formula one-to-one. There is a slight deviation, in that we multiply the result of the previous step with the sum of the current step, before we apply the division. The reason becomes obvious, when we look at $\binom{6}{3}$, for instance. According to the formula, we would compute $\frac{6+1-1=6}{1} \times \frac{6+1-2=5}{2} \times \dots$. The second factor, $\frac{5}{2}$, is not a natural number – we cannot express this value with the only tool we own so far. The result however is the same, which you can prove to yourself simply by completing the product above and comparing your result with the All-knowing Triangle. We will investigate binomial coefficients more deeply, especially the question why they always result in an integer in spite of division being involved.

4.8. Combinatorial Problems with Sets

Many real-world problems can be modelled in terms of *sets*. We have already used sets informally and, indeed, they are of tremendous importance in the whole field of mathematics – set theory is even believed to provide a sound fundamentation for most areas of mathematics. This, however, is strongly contested since more than hundred years now and today there are other candidates for this role besides set theory. But today many, if not most mathematicians, after long battles over the foundations of math mainly during the first half of the 20^{th}, are tired of discussing these issues.

Anyway, what is a set in the first place? Georg Cantor (1845 – 1918), one of the main inventors of set theory, provided several definitions, for instance: “a set is a Many that allows being thought of as a One” or: “a collection of distinct objects”. Both definitions are quite abstract, but, in this respect, they express a major aspect of set theory quite well.

The second definition, “collection of distinct objects”, serves our purposes well enough. A set can consist of any kind of objects, as long as these objects can be clearly distinguished. Examples are: The set of all green things, the set of all people, The set of Peter, Paul and Mary, the set of all animals that belong to the emperor, the set of the natural numbers from 1 to 9 the set of all natural numbers and so on.

There are different ways to define sets. We can first give a definition: the set of the natural numbers from 1 to 9, the set of all people, the members of the Simpsons family, *etc.* But we can also name the members explicitly: Homer, Marge, Bart, Lisa and Maggie or $1, 2, 3, 4, 5, 6, 7, 8, 9$.

The first way to define a set is called by *intension.* The intension of a set is what it implies, without referring explicitly to its members. The second way is called by *extension.* The extension of a set consists of all its members.

This distinction is used in different ways of defining lists in Haskell. One can define a list by extension: `[1,2,3,4,5]` or by intension: `[1..5]`, `[1..]`. A powerful tool to define lists by intension is list comprehension, for instance:

$[x \mid x \leftarrow [1\,..\,100], even\ x, x\ `mod`\ 3 \not\equiv 0]$,

which would contain all even numbers between 1 and 100 that are not multiples of 3.

Defining sets by intension is very powerful. The overhead of constructing a set by extension, *i.e.* by naming all its members, is quite heavy. If we had to mention all numbers we wanted to use in a program beforehand, the code would become incredibly large and we would need to work on it literally an eternity. Instead, we just define the kind of objects we want to work with. However, intension bears the risk of introducing some mind-boggling complications, one of which is infinite sets. For the time being, we will steer clear of any of these complications. We have sufficient work with the kind of math that comes without fierce creatures like infinity.

Sets, as you have already seen, are written in braces like, for instance: $\{1, 2, 3\}$. The members of a set, here the numbers 1, 2 and 3, are called elements of this set, it holds true, for example, that $1 \in \{1, 2, 3\}$ and $0 \notin \{1, 2, 3\}$.

A similar relation is *subset.* A set A is subset of another set B, iff all elements of A are also in B: $\{1\} \subseteq \{1, 2, 3\}$, $\{1, 3\} \subseteq \{1, 2, 3\}$ and also $\{1, 2, 3\} \subseteq \{1, 2, 3\}$. The last case is interesting, because it asserts that every set is subset of itself. To exclude this case and only talk about subsets that are smaller than the set in question, we refer to the *proper* or *strict subset*, denoted as $A \subset B$. An important detail of the subset relation is that there is one special set that is subset of any set, *viz.* the *empty set* $\{\}$ that does not contain any element and which is often denoted as $\varnothing$,

As you can see in the example $\{1, 2, 3\}$, a set may have many subsets. The set of all possible subsets of a set is called the *powerset* of this set, often written $P(S)$ for a set S. The powerset of $\{1, 2, 3\}$, for example, is: $\{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Does this remind you of something? Perhaps not yet. What if I was to ask: how many elements are there in the powerset of a set with n elements? Well, there is the empty set, the set itself, then sets with one element, sets with two elements and so on. How many sets with k elements are there in the powerset of a set with n elements? The answer is: there are as many sets of k elements as there are ways to select k items out of n. In other words, the size of the powerset equals the sum of all binomial coefficients $\binom{n}{k}$ for one specific value of n, *i.e.* the sum of all values in one row of Pascal's Triangle. For $n = 0$, n is the number of elements of the set, we have: $\binom{0}{0} = 1$, since the only subset of $\varnothing$ is $\varnothing$. For $n = 1$, we have: $\binom{1}{0} + \binom{1}{1} = 2$. For $n = 2$, we have: $\binom{2}{0} + \binom{2}{1} + \binom{2}{2}$, which is $1 + 2 + 1 = 4$. For $n = 3$, we have: $\binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3}$, which is $1 + 3 + 3 + 1 = 8$, for $n = 4$, we have: $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4}$, which is $1 + 4 + 6 + 4 + 1 = 16$.

Probably, you already see the pattern. For 5 elements, there are 32 possible subsets; for 6 elements, there are 64 subsets, for 7, there are 128 and for 8 there are 256 subsets. In general, for a set with n elements, there are 2^n subsets and, as you may confirm in the Triangle in the previous section, the sum of all binomial coefficients in one row of the Triangle is also 2^n. This, in its turn, implies that the sum of the coefficients in an expression of the form $a^n + \binom{n}{1}a^{n-1}b + \dots + \binom{n}{n-1}ab^{n-1} + b^n$, as well, is 2^n.

Is there a good algorithm to construct the powerset of a given set? There are in fact many ways to build the powerset, some more efficient or more elegant than others, but really *good* in the sense that it efficiently creates powersets of arbitrarily large sets is none of them. The size of the powerset increases exponentially in the size of the input set, which basically means that it is not feasible at all to create the powerset in most cases. The powerset of a set of 10 elements, for instance, has 1024 elements. That of a set of 15 elements has already 32 768 and a set of 20 elements has more than a million.

Here is a Haskell implementation of a quite elegant and simple algorithm:

```
ps :: (Eq a) ⇒ [a] → [[a]]
ps [] = [[]]
ps (x : xs) = ps xs ++ map (x:) (ps xs)
```

Note that we use lists instead of sets. There is a set module in Haskell, but since we will not work too much with sets, we stick to lists with the convention that there should be no duplicates in lists that represent sets.

Let us see how the *ps* function works for the input $\{1, 2, 3\}$. We start with *ps* $(1 : [2, 3])$

and immediately continue with *ps* $(2 : [3])$ and, in the next round, with *ps* $(3 : [\,])$, which then leads to the base case *ps* $[\,] = [[\,]]$. On the way back, we then have $[[\,]] +\!\!+$ *map* (3:) $[[\,]]$, which leads to the result $[[\,], [3]]$. This, one step further back, leads to $[[\,], [3]] +\!\!+$ *map* (2:) $[[\,], [3]]$, which results in $[[\,], [3], [2], [2, 3]]$. In the previous step, we then have: $[[\,], [3], [2], [2, 3]] +\!\!+$ *map* (1:) $[[\,], [3], [2], [2, 3]]$, resulting in $[[\,], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]$ and we are done.

A completely different approach is based on the observation that there are 2^n possible subsets of a set with n elements and this happens to be the number of values one can represent with a binary number of length n, namely the values $0 \dots n - 1$. Binary numbers, which we will discuss in more detail later, use only the digits 0 and 1 instead of the digits $0 \dots 9$ as we do with decimal numbers. In binary numbers we would count like

binary	decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Indeed, there are 10 kinds of people in the world: those who understand binary numbers and those who do not.

To construct the powerset of a set of n elements, we can use binary numbers with n digits. We would loop over this numbers starting from 0 and move up to the greatest number representable with n digits. For $n = 3$, we would loop through: 000, 001, 010, 011, 100, 101, 110, 111. Each of these numbers describes one subset of the input set, such that the k^{th} digit of each number would tell us, whether the k^{th} element of the input set is part of the current subset. The number 000 would indicate the empty set. The number 001 would indicate that the first element is in the set: $\{1\}$. The number 010 would indicate that the second element is in the set: $\{2\}$. The number 011 would indicate that the first and the second element are in the set: $\{1, 2\}$ and so on.

An important issue to gain any speed advantage by this scheme is how to map the binary numbers to elements in the input set. We could naïvely use an underlying representation of binary numbers like lists – as we have done for our natural numbers – iterate through these lists and, every time we find a 1, add the corresponding element of the input set to the current subset. But this would mean that we had to loop through 2^n lists of length n. That does not sound very efficient.

The key is to realise that we are talking about numbers. We do not need to represent binary numbers as lists at all. Instead, we can just use decimal numbers and extract the positions where, in the binary representation of each number, there is a 1.

To illustrate this, remember that the value of a decimal number is computed as a sum of powers of 10: $1024 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$. The representation of 1024 as powers of two is of course much simpler: $1024 = 1 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + \dots + 0 \times 2^0$ or, for short: $1024 = 2^{10}$. Let us look at a number with a simple decimal representation like 1000, which, in powers of 10, is simply: 10^3. Represented as powers of two, however: $2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3$, which is $512 + 256 + 128 + 64 + 32 + 8 = 1000$.

The point is that the exponents of 1000 represented as powers of two indicate where the binary representation of 1000 has a 1. 1000 in the binary system, indeed, is: 1111101000, whereas 1024 is 10000000000. Let us index these numbers, first 1000:

10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	1	0	0	0

and 1024:

10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0

You see that the indexes in the first row that have a 1 in the second row correspond to the exponents of the powers of two that sum up to the respective number, *i.e.* 9, 8, 7, 6, 5 and 3 for 1000 and 10 for 1024. We can, hence, interpret the exponents of the powers of two as indexes into the set for which we want to construct the powerset. Think of the input set as an array in a language like C, where we can refer to an element of the set directly by addressing the memory cell where is resides: `x = set[0];` for instance, would give us the first element of the set.

When we look at how a number, say d, is computed as sum of powers of two, we can derive the following algorithm: compute the greatest power of two that is smaller than or equal to d and than do the same with the difference between d and this power of two until the difference, which in analogy to division, we may call the remainder, is zero. The greatest power of two $\leq d$ is just the log base 2 of this number rounded down to the next natural number. A function implementing this in Haskell would be (cheating on our number type by using floating point numbers):

```
natLog :: Natural → Natural → (Natural, Natural)
natLog b n = let e = floor $ logBase (fromIntegral b)
```

```
                            (fromIntegral n)
             in (e, n − b ↑ e)
```

This function takes a natural number b, the base, and a natural number n. The result consists of two numbers (e, r) that shall fulfil the condition $n = b^e + r$.

We can use this function to obtain all exponents of the powers of two that sum up to a given number:

```
binExp :: Natural → [Natural]
binExp 0 = [ ]
binExp 1 = [0]
binExp n = let (e, r) = natLog 2 n in e : binExp r
```

That is, for input 0 and 1, we explicitly define the results [ ] and [0]. Here, [ ] means that there is no 1 in the binary representation and [0] means that there is 1 at the first position (indexed by 0). For any other number n, we calculate the exponent e of the greatest power of two $\leq n$ and the remainder r and add e to the list that will result from applying *binExp* to r. Let us look at the example 1000. We start with *natLog* 2 1000:

$\mathit{binExp}\ 1000 = (9, 1000 - 2^9 = 1000 - 512 = 488)$
$9 : \mathit{binExp}\ 488 = (8, 488 - 2^8 = 488 - 256 = 232)$
$9 : 8 : \mathit{binExp}\ 232 = (7, 232 - 2^7 = 232 - 128 = 104)$
$9 : 8 : 7 : \mathit{binExp}\ 104 = (6, 104 - 2^6 = 104 - 64 = 40)$
$9 : 8 : 7 : 6 : \mathit{binExp}\ 40 = (5, 40 - 2^5 = 40 - 32 = 8)$
$9 : 8 : 7 : 6 : 5 : \mathit{binExp}\ 8 = (3, 8 - 2^3 = 8 - 8 = 0)$
$9 : 8 : 7 : 6 : 5 : 3 : \mathit{binExp}\ 0 = [\,]$
$9 : 8 : 7 : 6 : 5 : 3 : [\,]$,

which, indeed, is the list of the exponents of the powers of two that add up to 1000.

Now we need a function that loops through all numbers $0 \dots 2^n - 1$, calculates the exponents of the powers of two for each number and then retrieves the elements in the input set that corresponds to the exponents:

```
ps2 :: (Eq a) ⇒ [a] → [[a]]
ps2 [ ] = [[ ]]
ps2 xs = go (2 ↑ (length xs) − 1) 0
  where go n i | i ≡ n      = [xs]
               | otherwise = let s = map exp2idx $ binExp i
                             in s : go n (i + 1)
        exp2idx x = xs !! (fromIntegral x)
```

The function *ps*2 returns just a set that contains the empty set when called with the empty set. Otherwise, it enters a loop with two parameters: $2^{(length\ xs)} - 1$, which is the greatest number that can be represented with a binary number with n digits, when we start to count at 0, and the number we start with, namely 0. For each number i: if we have reached the last number, we just know the corresponding subset is the input set itself. Otherwise, we map the function *exponent* → *index* to the result of *binExp* applied to the current number i. The mapping function, *exp2idx*, uses the list index operator `!!` to get the element of the input list xs at the position x, which is just an exponent. (Note that we have to convert x from $Natural$ to Int, since `!!` expects an Int value.)

This algorithm exploits a fascinating *isomorphism* – an analogous structure – between binary numbers and powersets. With an appropriate data structure to represent sets, like *Vector*, and, of course, a more efficient number representation than our humble natural numbers, the algorithm definitely beats the one we implemented as *ps*. Furthermore, this algorithm can be paralellised according to number ranges, which is not possible with the previous algorithm, since, there, results depend on inermediate results, such that each step builds on a predecessor.

Unfortunately, lists show very bad performance with random access such as indexing. Therefore, $ps2$ is slower than *ps*. But using Haskell vectors (implemented in module *Data.Vector*) and Integers instead of our $Natural$, $ps2$ is indeed faster. The changes, by the way, are minimal. Just compare the implementation of $ps2$ and *psv*:

```
psv :: (Eq a) ⇒ [a] → [[a]]
psv [] = [[]]
psv xs = let v = V.fromList xs
         in go v (2 ↑ (length xs) − 1) 0
   where go v n i | i ≡ n      = [xs]
                  | otherwise = let s = map (exp2idx v) (binExp i)
                                in s : go v n (i + 1)
         exp2idx v x = v ! (fromIntegral x)
```

The changes to the code of $ps2$ relate to the introduction of v, a vector created from xs by using the *fromList* function from the vector module, which is qualified as V.

In practical terms, however, the performance of the powerset function does not matter too much, since, as already said, it is not feasible to compute the powerset of large sets anyway. Nevertheless, problems related to subsets are quite common. An infamous example is the *set cover* problem.

The challenge in the set cover problem is to combine given subsets of a set A so that the combined subsets together equal A. This involves an operation on sets we have not yet discussed. Combining sets is formally called *union*: $A \cup B$. The union of two sets, A and B, contains all elements that are in A or B (or both), for example: $\{1,2,3\} \cup \{3,4,5\} = \{1,2,3,4,5\}$.

Two other important set operations are intersection and difference. The intersection of two sets A and B, $A \cap B$, contains all elements x, such that $x \in A$ and $x \in B$. To continue with the example used above: $\{1,2,3\} \cap \{3,4,5\} = \{3\}$. The intersection of the union of two sets with one of these sets is just that set, $(A \cup B) \cap A = A$: $(\{1,2,3\} \cup \{3,4,5\}) \cap \{1,2,3\} = \{1,2,3,4,5\} \cap \{1,2,3\} = \{1,2,3\}$.

The difference of two sets A and B, $A \setminus B$, contains all elements in A that are not in B, for example: $\{1,2,3\} \setminus \{3,4,5\} = \{1,2\}$. If B is a subset of A, then the difference $A \setminus B$ is called the *complement* of B in A.

Now let us model the three set operations union, intersection and difference with Haskell lists. The simplest case is difference, since, assuming that we always use lists without duplicates, we can just use the predefined list operator `\\`. Union is not too difficult either using the function *nub*, which removes duplicates from a list:

```
union :: (Eq a) ⇒ [a] → [a] → [a]
union a b = nub (a ++ b)
```

Using *nub* is necessary, since merging the two lists will introduce duplicates for any $x \in a$ and $x \in b$.

Intersect is slightly more difficult. We could implement intersect by means of *nub*; we used *nub* in *union* to remove the duplicates of exactly those elements that we want to have in intersect. The intersect, hence, could be implemented as $a +\!\!+\ b \setminus\!\setminus nub\ (a +\!\!+\ b)$. This would define the intersect as the difference of the concatenation of two lists and the union of these two lists. Have a look at the example $A = \{1,2,3\}$ and $B = \{3,4,5\}$:

$A +\!\!+\ B \setminus\!\setminus nub\ (A +\!\!+\ B) =$
$[1,2,3] +\!\!+\ [3,4,5] \setminus\!\setminus nub\ ([1,2,3] +\!\!+\ [3,4,5]) =$
$[1,2,3,3,4,5] \setminus\!\setminus nub\ ([1,2,3,3,4,5]) =$
$[1,2,3,3,4,5] \setminus\!\setminus [1,2,3,4,5] =$
$[3]$.

This implementation, however, is not very efficient. Preferable is the following one:

```
intersect :: (Eq a) ⇒ [a] → [a] → [a]
intersect [] _ = []
intersect _ [] = []
intersect (a : as) bs | a ∈ bs    = a : intersect as bs
                      | otherwise =     intersect as bs
```

We first define the intersection of the empty set with any other set as the empty set. (Note the similarity of the role of the ∅ in union and intersect with that of 0 in addition and multiplication!) For other cases, we start with the first element of the first list, a, and check if it is also in bs; if so, we add a to the result set and continue with *intersect* on the tail of the first list; otherwise, we continue without adding anything in this round.

We now can state the set cover problem more formally: We have a set U, called the *universe*, and a set $S = \{s_1, s_2, \dots, s_n\}$ of subsets of U, $s_1 \subseteq U, s_2 \subseteq U, \dots, s_n \subseteq U$, such that the union of all the sets in S equals U, $s_1 \cup s_2 \cup \dots \cup s_n = U$. What is the least expensive union of a subset of S that yields U?

Least expensive may be interpreted in different ways. In the pure mathematical sense, it usually means the smallest number of sets, but in real world problems, least expensive may refer to lowest cost, shortest time, fewest people involved, *etc.* The problem is in fact very common. It comes up in scheduling problems where the members of S represent sets of threads assigned to groups of processors; very typical are problems of team building where the sets in S represent teams of people with complementing skills; but there are also problems similar to the *travelling salesman* problem where the sets in S represent locations that must be visited during a round trip.

So, how many steps do we need to solve this problem? To find the optimal solution, we basically have to try out all combinations of subsets in S. For $S = \{a, b, c\}$, $\{a\}$ may be the best solution, $\{b\}$ may be, $\{c\}$, $\{a, b\}$, $\{a, c\}$, $\{b, c\}$ and, of course, $\{a, b, c\}$. As you should see, that are 2^n possibilities, *i.e.* the sum of all binomial coefficients $\binom{n}{k}$ where n is the size of S. That, as we know, is not feasible to compute with large S's. There are, however, solutions for specific problems using heuristics.

Heuristics are helpers in otherwise exponential search problems. In practice, heuristics may be derived from the concrete problem domain. With respect to the examples mentioned above, it is often obvious that we do not want to combine threads on one processor that better work in parallel; concerning problems with teams, we could exclude combinations of people who do not like each other or we may want to construct gender balanced teams. Such restrictions and insights can be used to drastically reduce the number of possible solutions and, thus, making computation feasible. But think, for instance, of a general purpose operating system that does not have any previous knowledge about the user tasks it should run. No real-world heuristics are available for the kernel to find an optimal balance.

There are purely mathematical heuristics that may come to aid in cases where the problem domain itself does not offer reasonable simplifications. For the set cover problem, a known heuristic that reduces computational complexity significantly, is to search for local optimums instead of the global optimum. That is, we do not try to find the solution that is the best compared with all other solutions, but, instead, we make optimal decisions in each round. For example, if we had the universe $U = \{1, 2, 3, 4, 5, 6\}$ and $S = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 4\}, \{3, 5\}, \{1, 6\}\}$, the optimal solution would be $\{\{1, 2, 4\}, \{3, 5\}, \{1, 6\}\}$. The key to find this solution is to realise that the second set in S, $\{1, 2, 4\}$, is the better choice compared to the first set $\{1, 2, 3\}$. But to actually realise that, we have to try all possible combinations of sets, which are 2^n and, hence, too many. An algorithm that does not go for the global optimum, but for local optimums, would just take the first set, because, in the moment of the decision, it is one of two equally good options and there is nothing that would hint to the fact that, with the second set, the overall outcome would be better. This *greedy* algorithm will consequently find only a suboptimal solution, *i.e.* $\{1,2,3\}$, $\{1,4\}$ or even $\{1,2,4\}$, $\{3,5\}$ and $\{1,6\}$. It, hence, needs one set more than the global optimum.

In many cases, local optimums are sufficient and feasible to compute. This should be motivation enough to try to implement a greedy solution for the set cover problem. The algorithm will in each step take the set that brings the greatest reduction in the distance between the current state and the universe. We, first, need some way to express this distance and an obvious notion for distance is just the size of the difference between the universe and another set:

```
dist :: (Eq a) ⇒ [a] → [a] → Int
dist a b = length (a \\ b)
```

Now, we need a function, say, *best* that uses *dist* to find the set in S with the least distance to the universe and another function that repeatedly finds the local minimum using *best*, until either all sets in S have been used or no set in S is able to reduce the distance to the universe anymore. Here are these functions:

```
greedySetCover :: (Eq a) ⇒ [a] → [[a]] → [[a]]
greedySetCover u s = loop (length u) [] s
  where loop _ _ []  = []
        loop m rs xs = let (m', p) = best m rs [] xs
                       in if m' < m
                            then p : loop m' (p `union` rs) (delete p xs)
                            else []
        best m _ p []       = (m, p)
        best m r p (x : xs) = let m' = dist u (x `union` r)
                              in if m' < m then best m' r x xs
                                           else best m  r p xs
```

The measure for the current optimum is the variable m used in *loop* and *best*. The whole algorithm starts with $m = length(u)$, which is the worst possible distance, *viz.* the distance between $\varnothing$ and the universe.

The second parameter passed to *loop*, rs, is the union of partial results. It is initially empty. The third parameter is the set of subsets we are working on starting with S. With an empty S, *loop* is just $\varnothing$. Otherwise, it uses *best* to get the local optimum, which is the tuple (m', p), where p is the best choice for the local optimum and m' the distance of this set to the universe. If $m' < m$, we actually have found a solution that improves on the current state and we continue adding p to the result set, which results from the recursion of *loop* with m' as current optimum, the union of p and the partial result rs and the current instance of S without p. Otherwise, the result is just the empty set.

The function *best* simply goes through all elements of the current instance of S. If *best* arrives at the end of the list, it just returns the previously identified optimum (m, p). Otherwise, for each element of the current set of subsets, it computes the distance and, should the current distance improve on the result, continues with this current optimum, if it does not, it continues with the old parameters.

The fact that we do not go back in the *loop* function to test other options, but always stick with a solution once it was found makes this algorithm *greedy*: It takes the money and runs. What is the speed-up we obtain with this apparently ugly strategy? One call of *best* passes through the whole list, which, initially, is S. *loop*, if *best* has found an optimum that improves on the old result, removes the corresponding element from the list and repeates the process. This time, *best* will go through a list of $n-1$ elements, where n is the size of S. If it finds a new mimimum again, the corresponding element is removed, and we get a list of $n-2$ elements. The process repeats, until *best* does not find a new optimum anymore. In the worst case, this is only after all elements in the list have been consumed. The maximum number of steps that must be processed, hence, is $n+n-1+n-2+\dots+1$ or simply the series $\sum_{k=1}^{n} k$, which, as we already know as the Little Gauss, is $\frac{n^2+n}{2}$. For a set S with 100 elements, we would need to consider 2^{100} possible cases to compute the global optimum, which is 1 267 650 600 228 229 401 496 703 205 376. With the local optimum, we can reduce this number to $\frac{100\times 101}{2} = 5050$ steps. For some cases, the local minimum is therefore the preferred solution.

4.9. Stirling Numbers

We saw that the number of all permutations of a set equals the factorial of the number of elements in that set. We also looked at the cycle notation where permutations are encoded as results of subsequent applications of this permutation; (1 2 5)(3 4), for instance, applied once to the sequence (1 2 3 4 5), would yield (2 5 4 3 1). It is now quite natural to ask – at least for a mathematician – how many permutations there are for a given number of orbits in cycle notation.

To answer this question, we first have to know how many different combinations of a given number of orbits we actually may have. The orbits in cycle notation, in fact, are *partitions* of a set. Partitions are non-empty, distinct subsets. The union of the partitions of a complete partitioning of a set is just the original set. The orbits of the permutation given above, for instance, (1 2 5)(3 4) can be seen as subsets that, obviously, are not empty and, since they have no element in common, are distinct. Their union $\{1, 2, 5\} \cup \{3, 4\}$, as you can easily verify, equals the original set $\{1, 2, 3, 4, 5\}$.

We could, hence, think in the lines of the powerset to generate partitions. We just leave out the empty set and, eventually, pick only groups of sets that, together, add up to the whole set. It is in fact somewhat more complicated. To illustrate that let us look at an algorithm that generates all possibilities to partition a set into two partitions:

$$twoPartitions :: (Eq\ a) \Rightarrow [a] \rightarrow [([a],[a])]$$

```
twoPartitions = fltr ∘ p2
  where p2 []       =  [([], [])]
        p2 (x : xs) =  [(x : a, b) | (a, b) ← p2 xs] ++
                       fltr [(a, x : b) | (a, b) ← p2 xs]
        fltr = filter (λp → ¬ (null (fst p) ∨
                                null (snd p)))
```

This innocent looking lines of code are quite tricky. The basic idea is implemented in *p2*: For an empty set, *p2* returns a pair of empty sets. For any other set, it applies *p2* twice on the *tail* of the list and adds the *head* once to the first of the pair and once to the second of the pair. This sounds easy, but there is an issue: The intermediate result sets will contain empty sets. In the first result, the second set is empty and, in the second result, the first one is empty. To solve this problem, we explicitly filter empty sets out (using *fltr*). If we applied the filter once on the overall result, we would get the following pairs for the set $\{1, 2, 3\}$:

$([1, 2], [3])$
$([1, 3], [2])$
$([1], [2, 3])$
$([2, 3], [1])$
$([2], [1, 3])$
$([3], [1, 2])$.

All results are correct, but there are too many of them. More specifically, some of the results are repeated. $([1, 2], [3])$ and $([3], [1, 2])$ are different tuples of course, but they describe the same partitioning consisting of the subsets $\{1, 2\}$ and $\{3\}$.

In the code above, this issue is solved by applying the filter once again, *viz.* on the second intermediate result. Let us look at how the second list comprehension develops. After the first application of *p2*, we have a tuple of two empty sets:

$p2\ (3 : [\,]) = [(a, 3 : b) \mid (a, b) \leftarrow [([\,], [\,])]]$
$p2\ (3 : [\,]) = [([\,], [3])]$.

This result is filtered out, because the first set is empty. In the next round we consequently have, as input, only the result of the first, unfiltered, list comprehension:

$p2\ (2 : [3]) = [(a, 2 : b) \mid (a, b) \leftarrow [([3], [\,])]]$
$p2\ (2 : [3]) = [([3], [2])]$,

which is preserved. We then get to the final round where the input is now the result of the first comprehension plus the one created above:

$p2\ (1 : [2, 3]) = [(a, 1 : b) \mid (a, b) \leftarrow [([3], [2]), ([2, 3], [\,])]$
$p2\ (1 : [2, 3]) = [([3], [1, 2]), ([2, 3], [1])]$.

The result of the first comprehension in the last round consists of the pairs: $([1,3],[2])$, which results from the input $([3],[2])$, and $([1,2,3],[\,])$, which is removed by the filter on the final result of *p2*. This gives the correct result:

$([1,3],[2])$
$([2,3],[1])$
$([3],[1,2])$.

As long as we create only two partitions, we can use pairs and the nice list comprehension to make the code clear. For the generation of k partitions, the code becomes somewhat more obscure. It is in particulare not sufficient anymore to call the filter twice. Instead, we need an additional function that removes the permutations of sets of partitions. To partition the set $\{1,2,3,4\}$ into three partitions, for example, we need to remove all but one of

$\{1,2\},\{3\},\{4\}$,
$\{1,2\},\{4\},\{3\}$,
$\{4\},\{1,2\},\{3\}$,
$\{3\},\{1,2\},\{4\}$,
$\{4\},\{3\},\{1,2\}$ and
$\{3\},\{4\},\{1,2\}$.

We will not develop this algorithm here, because it is highly inefficient. We would create much more sets than necessary and, then, we still have to generate permutations to remove those sets that are superfluous. Fortunately, there is an alternative very similar to that we have used to create powersets. For powersets, we used all binary numbers from 0 to $2^n - 1$, where n is the number of elements in the set. To generate partitions, we have to modify this idea in two respects.

First, we do not have the simple decision whether an element is in the current subset or not, but in which of k partitions it is. To get this information, we need a number system with the base k. For two partitions, this is just the binary number system. For three partitions, it would be a number system with base 3. For four partitions, we would use a number system with base 4 and so on.

Second, we have to restrict the numbers in a way that the result set points only to distinct partitionings. Imagine we want to know how to partition the set $\{1,2,3,4,5\}$ into three subsets. We would then need numbers of the form: 01200 or 01002. The first number, 01200, would point to the partitions $\{1,4,5\},\{2\},\{3\}$ and the second number, 01002, would point to the partitions $\{1,3,4\},\{2\},\{5\}$. In other words, the digits of the number indicate the partition in which the element at this position (in the original sequence) would be starting to count at index 0 for the first partition. Obviously, the number 01000 would be no good, because it does not describe a set of three partitions, but only one of two partitions. Also, with number 00012 already in the result, we do not want to generate the number 22210, because the corresponding sequences of partitions $\{1,2,3\},\{4\},\{5\}$ and $\{5\},\{4\},\{1,2,3\}$ are permutations of each other and, thus, describe the same set of partitions.

There is a simple trick to avoid such duplications and this trick has a name: *Restricted Growth Strings*, RGS for short. RGS are similar to numbers, but have leading zeros – they are therefore strings rather than proper numbers. The length of RGS depends on the purpose for which they are used. In our case, we want their length to equal the number of elements in the original set.

When counted up, RGS grow in an ordered fashion, such that lesser digits appear before greater ones. For instance, we allow numbers like 0012 and 0102, but do not allow such like 0201 or 1002. This implies that each new digit is at most one greater than the greatest digit already in the number, *i.e.* $a_i \leq 1 + \max a_1, a_2, ...a_{i-1}$. This restriction rules out numbers with a combination of digits that has already appeared with smaller RGS before. Of the strings 0123, 0132, 0213, 0231, 0312 and 0321 only the first is a valid RGS. With the others, either 3 or 2 appear before 1, violating the restriction that no digit must be greater than the greatest number appeared so far plus 1. You can easily verify that all those strings point to the same partitioning of set $\{1, 2, 3, 4\}$, namely permutations of the set $\{\{1\}, \{2\}, \{3\}, \{4\}\}$.

The ordered growth also implies that the first digit in an RGS is always 0. Otherwise, if 0 did not appear in the string at all, the first partition would be empty and the partitioning would, hence, be invalid; if 0 did appear later in the string, the string would not be ordered, *i.e.* a greater number would appear before the smallest possible number *zero*.

To be sure that the RGS-technique effectively avoids duplication of subset by suppressing permuations, we should at least sketch a proof of the concept. We should prove that restricted growth makes complementing groups of digits impossible, such that all digits k_1 and k_2 swap their positions from one string to the other. The following diagram shows four positions in a string where, at positions i and $i + 1$, there is the digit k_1 and, at positions j and $j + 1$, there is the digit k_2:

...	i	$i+1$	...	j	$j+1$	...
...	k_1	k_1	...	k_2	k_2	...

We assume that $j > i + 1$ and we assume that all occurences of k_1 and k_2 in the string are shown. In other words, this partial string shows the partitions $k_1 = \{i, i + 1\}$ and $k_2 = \{j, j + 1\}$.

We prove by contradiction on restricted growth and assume that this string is possible with both cases, $k_1 < k_2$ or, alternatively, $k_2 < k_1$. Consider the case $k_2 < k_1$. In this case k_2 must appear in a position $p < i$, otherwise, ordering would be violated. But this contradicts the assumption that $k_2 = \{j, j + 1\}$. So, either we violate ordering or k_2 is not shown completely in the diagram above and, then, the subsets are not complementing. Ordering is violated because it implies that any digit a_i in the string is at most $1 + \max a_1, a_2, \dots, a_{i-1}$. i is either 0, then k_1, per definition, is 0 as well and no (natural) number is less than 0, hence k_2 cannot be less than k_1; or i is not 0, then there must be a digit $k_0 = k_1 - 1$. If we assume that $k_2 < k_1$, we must assume that $k_1 - 1 < k_2 < k_1$ and, hence, that $0 < k2 < 1$. But that cannot be, since k_2 is still a natural number. □

To implement the RGS analogy, we first need a function that converts decimal numbers into numbers with base b. We have already looked at such functions, for $b = 2$ in the previous section, which we called *binExp*, and, for $b = 10$, in the previous chapter in the context of the conversion function *integer2Num*. *binExp* was tailored for binary numbers, since it yielded only the positions where the binary result would have a 1. That information is obviously not sufficient for number systems with $b > 2$, where the decision which number to put at a given position is not binary.

Let us recall how we converted integer to our natural number type. We divided the number by 10, collecting the remainders and continuing on the quotient, like in the following example:

$1000 \text{ `}quotRem\text{` } 10 = (100, 0)$
$100 \text{ `}quotRem\text{` } 10 = (10, 0)$
$10 \text{ `}quotRem\text{` } 10 = (1, 0)$
$1 \text{ `}quotRem\text{` } 10 = (0, 1).$

Now, the remainders of the subsequent divisions bottom-up would read $1, 0, 0, 0$, which are just the components of the decimal representation of the number 1000. If we do this with $b = 2$, we would see:

$1000 \text{ `}quotRem\text{` } 2 = (500, 0)$
$500 \text{ `}quotRem\text{` } 2 = (250, 0)$
$250 \text{ `}quotRem\text{` } 2 = (125, 0)$
$125 \text{ `}quotRem\text{` } 2 = (62, 1)$
$62 \text{ `}quotRem\text{` } 2 = (31, 0)$
$31 \text{ `}quotRem\text{` } 2 = (15, 1)$
$15 \text{ `}quotRem\text{` } 2 = (7, 1)$
$7 \text{ `}quotRem\text{` } 2 = (3, 1)$
$3 \text{ `}quotRem\text{` } 2 = (1, 1)$
$1 \text{ `}quotRem\text{` } 2 = (0, 1).$

1000, in the binary system, hence, is 1111101000, the same result we have already obtained in the previous section. We can implement this procedure in Haskell as:[2]

```
toBaseN :: Int → Int → [Int]
toBaseN b = reverse ∘ go
  where go x = case x ‘quotRem‘ b of
```

---

[2] We use *Int* instead of *Natural* here, because, in the following, we will need list functions like *length* or *take* quite often; with *Natural*, we would have to add a lot of conversions, which is much harder to read.

```
                    (0, r) → [r]
                    (q, r) → r : go q
```

The *go*-part of this function applied to base $b = 3$ and, say, 1024 would develop as follows:

```
go 1024 = 1024 `quotRem` 3
1 : go 341 = 341 `quotRem` 3
1 : 2 : go 113 = 113 `quotRem` 3
1 : 2 : 2 : go 37 = 37 `quotRem` 3
1 : 2 : 2 : 1 : go 12 = 12 `quotRem` 3
1 : 2 : 2 : 1 : 0 : go 4 = 4 `quotRem` 3
1 : 2 : 2 : 1 : 0 : 1 : go 1 = [1]
1 : 2 : 2 : 1 : 0 : 1 : 1,
```

which, reversed, is $[1, 1, 0, 1, 2, 2, 1]$ and the correct representation of 1000 in the ternary system.

Now we need some functions to convert the number given in the b-ary system into an RGS. First we fill the number with leading zeros until it has the desired size n:

```
rgs :: Int → Int → Int → [Int]
rgs b n i = let r = toBaseN b i
                d = n − length r
                p = if d > 0 then take d (repeat 0) else [ ]
            in if d < 0 then [ ] else p ++ r
```

Note that, if the length of the result of *toBaseN* exceeds n, the function yields the empty list. This, in fact, is an error case that could be handled explicitly. On the other hand, returning the empty list is a good enough indication for an error and we could check for this error in code using *rgs* later.

We now define a wrapper around this conversion function to apply the restrictions:

```
toRgs :: ([Int] → Bool) →
  Int → Int → Int → (Int, [Int])
toRgs rst b n i = go (rgs b n i)
  where go r | ¬ (rst r)  = toRgs rst b n (i + 1)
             | otherwise = (i, r)
```

This function converts a decimal number to an RGS and checks if the result obeys the restrictions, which are passed in as a boolean function. If it does not, then the input is incremented by one and the function is called again. Otherwise, the function yields a tuple consisting of the decimal number that we have eventually reached and the RGS.

We define the growth restriction as follows:

```
rGrowth :: [Int] → Bool
rGrowth  []       = True
rGrowth  (x : xs) = go x xs
  where go _ []       = True
        go d (z : zs) = if z − d > 1 then False
                        else let d′ = max d z in go d′ zs
```

Since we want to see only partitionings with b subsets, we, still, need another restriction. We do not want to see RGS of the form 01000, when we ask for three partitions, or 01230, when we ask for five. For this end, we need the restriction that there must be b different digits in the resulting RGS. The restriction is easily implemented as:

```
hasN :: Int → [Int] → Bool
hasN b r = length (nub r) ≡ b
```

We apply this restrictions in yet another wrapper to call *toRgs*:

```
toRgsN :: Int → Int → Int → (Int, [Int])
toRgsN b = toRgs rst b
  where rst r = rGrowth r ∧ hasN b r
```

Finally, we can implement a loop that counts RGS up from 1 to the last number with leading 0:

```
countRgs :: Int → Int → [[Int]]
countRgs 1 n  = [rgs 1 n 1]
countRgs b n  | b ≡ n       = [[1 . . n − 1]]
              | b > n       = []
              | otherwise = go 1
  where go i = let (j, r) = toRgsN b n i
               in if head r ≢ 0 then [] else r : go (j + 1)
```

Note that we jump over numbers that do not obey the restrictions: we continue always with *go* applied to j, the first return value of *toRgsN*. j does not necessarily equal i; it depends on how many numbers have been ignored by *toRgs* because they did not obey the restrictions. When we call countRgs on 3, the numbers of partitions we want to have, and 4, the number of elements in the original set, we get the following RGS:

$[0, 0, 1, 2]$
$[0, 1, 0, 2]$
$[0, 1, 1, 2]$
$[0, 1, 2, 0]$
$[0, 1, 2, 1]$
$[0, 1, 2, 2]$,

which correspond to the partitionings of $\{1, 2, 3, 4\}$:

$\{1, 2\}, \{3\}, \{4\}$
$\{1, 3\}, \{2\}, \{4\}$
$\{1\}, \{2, 3\}, \{4\}$
$\{1, 4\}, \{2\}, \{4\}$
$\{1\}, \{2, 4\}, \{3\}$
$\{1\}, \{2\}, \{3, 4\}$.

This analogy between RGS and partitions is implemented as:

```
rgs2set :: (Eq a) ⇒ [Int] → [a] → [[a]] → [[a]]
rgs2set [] _ ps             = ps
rgs2set _ [] ps             = ps
rgs2set (r : rs) (x : xs) ps = rgs2set rs xs (ins r x ps)
  where ins _ _ []          = ⊥
        ins 0 p (z : zs)    = (p : z) : zs
        ins i p (z : zs)    = z : ins (i − 1) p zs
```

The function receives three arguments: The RGS, the original set we want to partition and the result set, which initially should contain *k* empty lists with *k* the number of partitions we want to obtain. (This pre-condition, actually, is not enforced in this code.) Note that the logic of using *k* empty lists is very similar to the trick we used in *twoParatitions* above.

When we have exhausted either the RGS or the original set, the result is just *ps*, the result set we passed in. Otherwise, we recurse with the tails of the RGS and the set (this way establishing the analogy) inserting the element of the set that corresponds to the current position of the RGS to the partition that, in its turn, corresponds to the digit of the RGS at this position. If the digit is 0, we just insert the element into the first list, otherwise we recurse (on *ins*) decrementing the digit by 1. Note that *ins* is ⊥ for the case that the result set is empty before we reach 0. This, obviously, would hint to an erroneous RGS and, hence, to a coding error.

We now can put everything together:

```
nPartitions :: (Eq a) ⇒ Int → [a] → [[[a]]]
nPartitions _ [] = []
nPartitions 1 xs = [[xs]]
nPartitions k xs | k ⩾ length xs = [[[x] | x ← xs]]
                 | otherwise     = go (countRgs k (length xs))
  where go []       = []
        go (r : rs) = rgs2set r xs (take k (repeat [])) : go rs
```

The function *nPartitions* receives the number of partitions we would like to have, *k*, and the original set, *xs*. If the set is empty, the result is empty, too. If we just want one partition, we return the set as its only partition. If k equals or exceeds the size of the set, we just return each element in its own set. (We could return an error for the case that k exceeds the size of the set, but, for sake of simplicity, we allow this case, returning an incomplete result.) Otherwise, we call *countRgs* and apply *rgs2set* to all elements of the result. The set of empty lists we need to start with *rgs2set* is created by *repeat* $[\,]$ which creates an infinite list containing the empty list – $[[\,],[\,],[\,],...]$ – from which we just take k, *i.e.* the number of partitions. The call *nPartitions* $2\ [1,2,3]$ would yield the expected result of all possibilities to partition $[1,2,3]$ in 2 subsets:

$[2,1],[3]$
$[3,1],[2]$
$[1],[3,2]$.

This result corresponds to the RGS:

001,
010 and
011.

The order of elements within partitions is explained by the fact that *ins* (in *rgs2set*) adds new elements using ":" – the element that was first inserted into the list is therefore the last one in the resulting partition.

It should be noted that, as for the powerset, we can optimise this code by using other data structures than lists. Since, as for the powerset as well, the number of possible partitionings of huge sets is incredibly large, it is not feasible to computate all partitionings of great sets anyway. We, therefore, leave it with a non-optimal implementation.

Now, that we have arrived here, the mathematically natural question occurs of how many partitions there are for a set of size n. Well, we have a tool to try that out. The following – *er* – triangle shows results of calls of *nPartitions*. Each line corresponds to the call $[\mathit{length}\ (\mathit{nPartitions}\ k\ [1\,..\,n]) \mid k \leftarrow [1\,..\,n]]$, where n starts with 1 at the top of the triangle and is counted up until 7 at its bottom.

1 1
2 1 1
3 1 3 1
4 1 7 6 1
5 1 15 25 10 1
6 1 31 90 65 15 1
7 1 63 301 350 140 21 1

On the first sight, the values in this triangle besides the ones in the outer diagonals appear less regular than those in Pascal's nice and tidy triangle. Nevertheless, already the second sight reveals some curious relations. The second diagonal from top-right to left-bottom, which reads 1,3,7,15,31,63, corresponds to the values $2^n - 1$. The second diagonal from top-left to right-bottom shows other numbers we already know, namely

1,3,6,10,15,21. If you take the differences, you will observe that each number is the sum of n and its predecessor. In other words, this diagonal contains the sum of all numbers from 1 to n, where n is the line number.

The triangle overall shows the *Stirling set numbers*, also known as the *Stirling numbers of the second kind*, which are denoted by

$$S(n,k) = \left\{ n \atop k \right\}. \tag{4.45}$$

The formula to compute Stirling numbers remarkably resembles Pascal's rule:

$$\left\{ n \atop k \right\} = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ k \times \left\{ n-1 \atop k \right\} + \left\{ n-1 \atop k-1 \right\} & \text{otherwise.} \end{cases} \tag{4.46}$$

This translates into Haskell as:

```
stirling2 :: Natural → Natural → Natural
stirling2 0 _ = 0
stirling2 _ 0 = 0
stirling2 1 1 = 1
stirling2 n k | k > n       = 0
              | otherwise = k ∗ (stirling2 (n − 1) k) +
                                (stirling2 (n − 1) (k − 1))
```

The code is almost a one-to-one translation of the mathematical formulation. However, there is an extra line, namely the base case *stirling2* _ 0 = 0. This base case must be introduced to avoid that we go below zero for k with the rule part *stirling2* $(n-1)$ $(k-1)$. The natural numbers are not defined for the range less than zero and so we have to stop here explicitly.

The sum of all Stirling numbers of the same row, *i.e.* $\sum_{k=1}^{n} \left\{ n \atop k \right\}$, is called the *Bell number* of n. The first 7 Bell numbers are: 1, 2, 5, 15, 52, 203, 877.

Since Stirling numbers of the second kind count the ways to partition a set into a given number of subsets, Bell numbers indicate all ways to partition a set, not restricting the number of subsets we want to obtain. The following table shows this relation for the set $\{1, 2, 3, 4\}$:

$\left\{ {4 \atop 1} \right\} = 1$	$\left\{ {4 \atop 2} \right\} = 7$	$\left\{ {4 \atop 3} \right\} = 6$	$\left\{ {4 \atop 4} \right\} = 1$
$\{1,2,3,4\}$	$\{1,2,3\},\{4\}$	$\{1,2\},\{3\},\{4\}$	$\{1\},\{2\},\{3\},\{4\}$
	$\{1,2,4\},\{3\}$	$\{1,3\},\{2\},\{4\}$	
	$\{1,3,4\},\{2\}$	$\{1,4\},\{2\},\{3\}$	
	$\{2,3,4\},\{1\}$	$\{1\},\{2,3\},\{4\}$	
	$\{1,2\},\{3,4\}$	$\{1\},\{2,4\},\{3\}$	
	$\{1,3\},\{2,4\}$	$\{1\},\{2\},\{3,4\}$	
	$\{1,4\},\{2,3\}$		

If you count the partitionings in one column, you get the Stirling number for that column; the number of all partitions in all columns equals the Bell number of 4, which is 15.

The inventors – or discoverers, depending on your philosophical view – of Bell numbers and Stirling numbers are two very interesting characters. Eric Temple Bell (1883 – 1960) was professor of mathematics in the United States for most of his life. But he was also an early science fiction writer and a math historian. His science fiction reached a higher level of science than most other publications in this genre at his time, but was often critised as poorly written and, in particular, for its weak characterisation of protagonists. His contributions to math history were even more fiercely critised as fictitious and romantic (as in the case of his biographical sketch of Évarist Galois) or as stereotypical (as in the case of his description of the life of Georg Cantor).

James Stirling (1692 – 1770) was from Scotland. He studied and taught in Oxford for some years, but had to flee from England, when he was accused of conspiracy based on his correspondence with Jacobites, that is supporters of the catholic kings, in particular James II who was deposed in 1688. After ten years of exile in Venice, he started to fear for his life again, because he discovered a trade secret of the glassmakers of Venice and returned to England with the help of his friend Isaac Newton. Much of Stirling's work is in fact tightly coupled with that of Newton. Stirling very much promoted Newton's discoveries and methods, for instance in his book *Methodus differentialis*. During the last years of his life, he was manager of the Scots Mining Company. During this period, he published mainly on topics of applied mathematics.

But let us return to the intial question. We were investigating the possible permutations with a given number of orbits in the cycle notation and have just learnt how to generate all possible k orbits of a set with n elements. We have found out how to partition a set of n elements into k subsets. However, the distinct subsets are not yet sufficient to generate all possible permutations, since the permutation of (1 2 3 4 5) $\sigma 1 = (1\ 2\ 5)(3\ 4)$ is not the same as $\sigma 2 = (1\ 5\ 2)(3\ 4)$:

$$\sigma 1 = 2\ 5\ 4\ 3\ 1$$

$$\sigma 2 = 5\ 1\ 4\ 3\ 2$$

So, do we need all permutations of the subsets? (Was our survey of RGS in vain?) Apparently not, since $\sigma 1$ is just the same permutation as $\sigma 3 = (1\ 2\ 5)(4\ 3)$. It is also the same as (5 1 2)(3 4). In fact, the cycle notation is indifferent concerning the starting point – for this reason, it is called cyclic. Therefore, not all permutations are relevant, but only those that change the order after the first element. An orbit permutating function aware of this peculiarity is:

```
permOrbits :: (Eq a) ⇒ Perm a → [Perm a]
permOrbits []       = [[]]
permOrbits (o : oo) = concat [map (:x) (oPerms o) | x ← permOrbits oo]
  where oPerms []       = []
        oPerms (x : xs) = [x : ps | ps ← perms xs]
```

This function just passes through all orbits of the input permutation creating permutations of each one using *oPerms*. It looks a bit weird that we do not just use *map* to create that result, but a list comprehension to which we even further apply *concat*. However, *map* does not yield the desired result. *map* would just create a list of permutated orbits – but we want to obtain complete cycles each of which may consist of more than one orbit. For this reason, we create permutations of the head and insert all permutations of the head to all results of the recursion of *permOrbits*.

The function we use for creating permutations of orbits is *oPerms*, which applies *perms*, all permutations of a list, to the tail of the input list. The head of the list, hence, remains always the same. For instance, *oPerms* $[3,4]$ is just $[3,4]$. *oPerms* $[1,2,5]$, however, yields $[1,2,5]$ and $[1,5,2]$.

Now, for one possible cycle, we can just apply all permutations resulting from *permOrbits*:

```
permsOfCycle :: (Eq a) ⇒ Perm a → [a] → [[a]]
permsOfCycle os xs = [permute o xs | o ← permOrbits os]
```

This function creates all permutations that are possible given one partitioning of the input set. We now map this function on all possible partitionings with k subsets:

```
permsWithCycles :: (Eq a) ⇒ Int → [a] → [[a]]
permsWithCycles k xs = concat [
  permsOfCycle x xs | x ← nPartitions k xs]
```

Applied on sets with n elements, for $n = 1 \dots 7$, and $k = 1 \dots n$, *permsWithCycles* yields results of length:

1	1						
2	1	1					
3	2	3	1				
4	6	11	6	1			
5	24	50	35	10	1		
6	120	274	225	85	15	1	
7	720	1764	1624	735	175	21	1

These, as you may have guessed already, are the Stirling numbers *of the first kind*, also known as *Stirling cycle numbers*, since they count the number of possible permutations with a given number of orbits in the cycle notation. They are denoted by

$$s(n,k) = \left[\begin{matrix} n \\ k \end{matrix}\right] \tag{4.47}$$

and can be calculated as

$$\left[\begin{matrix} n \\ k \end{matrix}\right] = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ (n-1) \times \genfrac{[}{]}{0pt}{}{n-1}{k} + \genfrac{[}{]}{0pt}{}{n-1}{k-1} & \text{otherwise.} \end{cases} \tag{4.48}$$

In Haskell, this would be:

```
stirling1 :: Natural → Natural → Natural
stirling1 0 _ = 0
stirling1 _ 0 = 0
stirling1 1 1 = 1
stirling1 n k | k > n     = 0
              | otherwise = (n − 1) ∗ (stirling1 (n − 1) k) +
                                     (stirling1 (n − 1) (k − 1))
```

We have seen that the sum of all Stirling numbers of the second kind in one row, *i.e.* $\sum_{k=1}^{n} \genfrac{\{}{\}}{0pt}{}{n}{k}$, is the Bell number of n. Can you guess what this sum is for Stirling numbers of the first kind?

Remember that each Stirling number of the form $\genfrac{[}{]}{0pt}{}{n}{k}$ shows the number of permutations with a given number of orbits in cycle notation. If you add up all possible permutations of all numbers of orbits $1 \dots n$, what do you get? Let us see:

For $n = 1$, we trivially get 1.
For $n = 2$, we get $1 + 1 = 2$.
For $n = 3$, we get $2 + 3 + 1 = 6$.
For $n = 4$, we get $6 + 11 + 6 + 1 = 24$.
For $n = 5$, we get $24 + 50 + 35 + 10 + 1 = 120$.

We know these numbers: these are the factorials of $n = n!$. Since the factorial counts the number of all possible permutations of a set, it is just natural that the Stirling numbers of the first kind, which count the possible permutations of a set with a given number of orbits, add up to the number of all possible permutations, *i.e.* the factorial of the size of the set. The triangle itself hints to that. The outer left diagonal, actually, shows the factorials! It shows the factorials of $n - 1$ though (with n indicating the row). If you think of how we create permutations of orbits – *viz.* as permutations of the tail of the orbit, without touching the head – it becomes immediately clear why the Stirling number $\left[{n \atop 1}\right]$, the one with only one partition, equals $(n - 1)!$.

There is still a lot to say about Stirling numbers. But that may involve concepts we have not yet discussed. So, we will have to come back to this topic later.

4.10. Eulerian Numbers

When we discussed Haskell, we studied a beautiful sorting algorithm, Hoare's *quicksort*. We already mentioned that *quicksort* is not the best algorithm in practice and it is this question to which we are coming back now.

The reason why *quicksort* is not optimal is that it looks at the world in black and white: a sequence is either sorted or it is not. But reality is not like this. Complete order, that is to say, a sequence where every element is at its place according to the order of the data type, starting with the least element in the sequence and each element being greater than its predecessor, is very rare, of course, and usually an effort is necessary to create complete order in this sense. Complete disorder, however, that is a sequence where no element is at its place, is quite rare too. In fact, it is as rare as complete order: for any sequence of unique elements, there is exactly one permutation showing complete order and one permutation showing complete disorder. Order and disorder balance each other.

For instance, the sequence $1, 2, 3, 4, 5$, is completely ordered; the permutation showing complete disorder would be $5, 4, 3, 2, 1$ and that is the original sequence reversed obeying as such another kind of order, *viz.* $\leq$ instead of $\geq$. All other permutations are in between. That is they show some order with a leaning either to the original sequence or to its reverse. For example: $5, 2, 4, 3, 1$ is close to the opposite order; reversed, it would be $1, 3, 4, 2, 5$ and close to order.

A sorting algorithm that exploits this pre-existing order in any input is *mergesort*. The underlying idea of *mergesort* is to merge two ordered lists. This can be implemented in Haskell simply as

```
merge :: (Ord a) ⇒ [a] → [a] → [a]
merge [] xs = xs
merge xs [] = xs
merge (x : xs) (y : ys) | x ⩽ y     = x : merge xs (y : ys)
                        | otherwise = y : merge (x : xs) ys
```

We first treat the base cases where one of the lists is empty, just yielding the respective other list. Then we compare the heads of the lists. The smaller one goes to the head of the merged list and we recurse with what remains.

We now want to use *merge* on an unordered, that is to say, incompletely ordered list. Therefore, we split the input into a list of ordered sublists. Ordered sublists are often referred to as *run*s. The positions where one ordered list ends and another one begins are called *stepdown*s, reflecting the idea that the sequence of increasing elements is interrupted by stepping down to a smaller element. Here is a Haskell function that splits a list into runs:

```
runs :: (Ord a) ⇒ [a] → [[a]]
runs []   = []
runs xs   = let (r, zs) = run xs in r : runs zs
  where run []  = ([], [])
        run [x] = ([x], [])
        run (x : y : zs) | x > y     = ([x], y : zs)
                         | otherwise = let (r, ys) = run (y : zs) in (x : r, ys)
```

The function applied to the empty list just yields the empty list. Applied to a non-empty list is calls the helper function *run* that returns a tuple consisting of two lists of as. The first element is then the head of the list resulting from *runs* applied to the second list.

The helper function *run* applied on the empty list yields a tuple of twice the empty list. Applied to a list containing only one element, it yield a tuple containing this list and the empty list. Otherwise, the first two elements of the list, x and y, are compared. If $x > y$, x goes to the first list of the resulting tuple, y goes to the rest of the list. This is a stepdown: the first element x is greater than its successor y and, hence, the natural order of the sequence is violated. Otherwise, we continue with $y : zs$ and insert x as the head of the resulting r, the first of the tuple. This is the case, where the run continues.

Let us look at an example: the permutation we already used above $1, 3, 4, 2, 5$. When we call *run* for the first time, we have:

run $(1 : 3 : [4, 2, 5])$

and we enter the *otherwise* alternative with $x = 1$:

run $(3 : 4 : [2, 5])$.

We again enter *otherwise*, this time $x = 3$:

run $(4:2:[5])$.

But this time we have $4 > 2$ and enter the first branch, that is we yield

$([4], 2:[5])$.

Going backwards, we see:

$(3:[4],[2,5])$
$(1:[3,4],[2,5])$,

which finally appears as result of the first call to *run* in *runs*, which leads to $[1,3,4]:$ *runs* $[2,5]$. $[2,5]$ is now handled in the same way and we obtain the overall result $[[1,3,4],[2,5]]$. This way, thee entire list is split into two runs.

When we look at the other example, the permutation that was closer to disorder, there are more runs: $5|24|3|1$, which is the proper mathematical notation for the list $[[5],[2,4],[3],[1]]$ consisting of four runs. If we reverse the list, however, the number of runs reduces and we get the list with only two runs above.

Can we exploit the fact that we can reduce the number of runs by reverting the list? Yes, of course, otherwise the question would not have been asked. It turns out that, for lists with unique elements, if the number of runs r is greater than $\frac{n}{2}+1$, then r', the number of runs of the reversed list is less than r. In other words, we could check the number of runs, before we start to sort, and revert the list if the number of runs is greater than the half of the length of the list plus 1.

Note that in the real world we often see lists with repeated elements. The repetitions, however, would not spoil the result, since repetitions would just reduce the possible number of runs. In consequence, it may happen that reverting the list, even though the number of runs is less than the half of the list size plus 1, would improve performance. But without reverting, the algorithm is still good, *viz.* comparable to the performance of a cousin of the same size without repetitions.

Let us look at an implementation of *mergesort* that exploits order in the input in this sense. First, we need a function that merges the runs, that is a merge for a list of lists. Let us call this function *multimerge*:

```
multimerge :: (Ord a) ⇒ [[a]] → [a]
multimerge []          = []
multimerge [xs]        = xs
multimerge (x : y : zs) = merge (merge x y) (multimerge zs)
```

The function reduces a list of lists of a to a plain list of a. For the empty list, it yields the empty list. For a list that contains only one single list, it yields just that list. For a list with more elements, it merges the first two lists and merges the resulting list with the list that results from *multimerging* the rest. With an example that will become clearer. But let us not use the list with two runs, because that would not let us see the recursion on *multimerge*. Instead, we use the non-optimal list with four runs. We would start with:

multimerge [[5] : [2, 4] : [[3], [1]]

and perform

merge (*merge* [5] [2, 4]) (*multimerge* [[3], [1]]),

which is

merge [2, 4, 5] (*multimerge* [[3], [1]])

and results in the call to *multimerge*:

multimerge [[3], [1]].

This reduces to

merge (*merge* [3] [1]) (*multimerge* [ ]),

which is

merge [1, 3] [ ],

which, in its turn, is [1, 3]. Going backwards, we obtain

merge [2, 4, 5] [1, 3],

which is [1, 2, 3, 4, 5].

We can now put everything together:

```
mergesort :: (Ord a) ⇒ [a] → [a]
mergesort l = let  rs = runs l
                   n′ = length l
                   n  = if even n′ then n′ else n′ + 1
              in if length rs > (n ‘div‘ 2) + 1
                 then multimerge $ runs (reverse l)
                 else multimerge rs
```

We first create the runs for the input list l. We then compute the length of l n'. If n' is not even, we add one, just to be sure, we later refer to at least the half of n'. Then, if the length of the runs is more than half of n plus 1, then we run *multimerge* on the runs of the reversed input list, otherwise, we run *multimerge* on the runs of l.

There is a well-known mathematical concept that is closely related to the conept of runs: the *Eulerian numbers*, which, as the Stirling numbers, come in two flavours ingeniously called Eulerian numbers of the first kind and Eulerian numbers of the second kind.

The Eulerian numbers of the first kind, denoted by $\left\langle {n \atop m} \right\rangle$, count the number of permutations of a set of n distinct elements where m numbers are greater than the previous element. They, hence, do not count the number of runs directly. They, first, use the number of elements that are actually member of a run without the first element of that run (which is a stepdown). This, actually, is the value of m, which counts, in other words, how many of the elements are not stepdowns. They, second, count the number of permutations that have m elements in such a configuration. For instance, the set $\{1, 2, 3, 4, 5\}$ has $5! = 120$ permutations. There is exactly one permutation where 4 elements are greater than their predecessor, namely the permutation $1, 2, 3, 4, 5$. Hence: $\left\langle {5 \atop 4} \right\rangle = 1$. There is also only one permutation with no element greater than its predecessor, namely $5, 4, 3, 2, 1$. Hence: $\left\langle {5 \atop 0} \right\rangle = 1$. There are 26 permutations with only 3 elements greater than their predecessor and 66 with only 2 elements greater than their predecessor.

You perhaps see immediately that Eulerian numbers can be used to compute the average running time of *mergesort* for input of a given size n. The Eulerian numbers allow us to compute the probability of that input having 1 run, 2 runs, ..., n runs. For the example above, we have the probabiliy $\frac{1}{5!=120}$ that there is no stepdown at all and that we, hence, do not have to do anything; we have the probability $\frac{26}{120} = \frac{13}{60}$ that we have to do only one merge step and the probability of $\frac{66}{120} = \frac{11}{20}$ that we have to do two merge steps and so on. (Knuth provides an extensive analysis in the third volume of his masterpiece.)

Let us look at a smaller set, where we can actually look at all permutations, say $\{1, 2, 3\}$. There is of course 1 permutation for 2 non-stepdowns and also 1 for no non-stepdown at all: $\left\langle {3 \atop 0} \right\rangle = \left\langle {3 \atop 2} \right\rangle = 1$. For 1 element greater than its predecessor, there are $\left\langle {3 \atop 1} \right\rangle = 4$ possibilities, namley: $213, 231, 132, 312$. The sum of all values is of course the number of all permutations, *i.e.* $3! = 6$: $\left\langle {3 \atop 0} \right\rangle + \left\langle {3 \atop 1} \right\rangle + \left\langle {3 \atop 2} \right\rangle = 1 + 4 + 1 = 6$.

Here are some values arranged in – oh, you guessed it already:

1	1						
2	1	1					
3	1	4	1				
4	1	11	11	1			
5	1	26	66	26	1		
6	1	57	302	302	57	1	
7	1	120	1191	2416	1191	120	1

Make sure yourself that, for each line n, the following identity holds:

$$n! = \sum_{m=0}^{n-1} \left\langle {n \atop m} \right\rangle \tag{4.49}$$

There is a recursive formula to compute the Eulerian numbers, which is

$$\genfrac{\langle}{\rangle}{0pt}{}{n}{m} = (n-m)\genfrac{\langle}{\rangle}{0pt}{}{n-1}{m-1} + (m+1)\genfrac{\langle}{\rangle}{0pt}{}{n-1}{m}, \tag{4.50}$$

with $\genfrac{\langle}{\rangle}{0pt}{1}{0}{m} = \genfrac{\langle}{\rangle}{0pt}{1}{n}{0} = \genfrac{\langle}{\rangle}{0pt}{1}{n}{n-1} = 1$. In Haskell, this can be implemented as:

```
eulerian1 :: Natural → Natural → Natural
eulerian1 0 _   = 1
eulerian1 _ 0   = 1
eulerian1 n m | m ≡ n − 1 = 1
              | otherwise  = (n − m) ∗ eulerian1 (n − 1) (m − 1)
                           + (m + 1) ∗ eulerian1 (n − 1) m
```

There is also a closed form to compute the nth Eulerian number and this closed form reveals a relation of the Eulerian numbers with the binomial coefficients:

$$\genfrac{\langle}{\rangle}{0pt}{}{n}{m} = \sum_{k=0}^{m} (-1)^k \binom{n+1}{k} (m+1-k)^n. \tag{4.51}$$

This expands into a series where the results of the products $\binom{n+1}{k}(m+1-k)^n$ are alternately added or subtracted. For instance for $\genfrac{\langle}{\rangle}{0pt}{1}{5}{2}$:

$$\genfrac{\langle}{\rangle}{0pt}{}{5}{2} = (-1)^0 \times \binom{6}{0} \times (2+1-0)^5 + (-1)^1 \times \binom{6}{1} \times (2+1-1)^5 + (-1)^2 \times \binom{6}{2} \times (2+1-2)^5, \tag{4.52}$$

which is

$$\begin{array}{cccccccc}
 & 1 & \times & 3^5 & & & = & 243 \\
- & 6 & \times & 2^5 & = & 6 \times 32 & = & 192 \\
+ & 15 & \times & 1^5 & & & = & 15,
\end{array}$$

hence: $243 - 192 + 15 = 66$.

It perhaps helps to get the closed form right to look at it in Haskell:

```
eu1closed :: Natural → Natural → Natural
eu1closed n m = go 0
  where go k = let a = (−1) ↑ k
                   b = choose (n + 1) k
                   c = (m + 1 − k) ↑ n
               in (a ∗ b ∗ c) + if k ≡ m then 0
                                         else go (k + 1)
```

The Eulerian numbers of the second kind are as well quite interesting. They deal with *multisets*, that is sets with repeated elements and, as such, they may pave the way to a theory for real world input. In concrete, they count the number of permutations of a multiset with n different elements with m ascents, where an ascent occurs whenever a number is greater than its predecessor. The concept of an ascent, hence, is stronger than that of a non-stepdown. A non-stepdown would include a number that equals its predecessor, but that is not an ascent.

To understand Eulerian numbers of the second kind, we first have to understand how many permutations there are for multisets. It is not the factorial, but the *doublefactorial*, also called the *semifactorial*, of $2n - 1$, where n is the number of unique elements in the multiset. For instance, the multiset $\{1, 1, 2, 2, 3, 3\}$ has $n = 3$ unique elements, namely 1, 2 and 3. The doublefactorial of $2n - 1 = 5$, denoted by $n!!$, is 15.

The doublefactorial of n is the product of all numbers $1 \dots n$ that have the same parity as n. If n is even, we multiply all even numbers $2 \dots n$, otherwise, if n is odd, we multiply the odd numbers $1 \dots n$. In Haskell, this may look like this:

```
doublefac :: Natural → Natural
doublefac 0 = 1
doublefac 1 = 1
doublefac n = n * doublefac (n − 2)
```

For instance, $5!! = 1 \times 3 \times 5 = 15$ and $6!! = 2 \times 4 \times 6 = 48$.

The permutations of the multiset $\{1, 1, 2, 2, 3, 3\}$ are

332211,
221133, 221331, 223311, 233211, 113322, 133221, 331122, 331221,
112233, 122133, 112332, 123321, 133122, 122331.

In the first line, there is one permutation with no ascent, in the second line, there are 8 permutations with one ascent and in the third line, there are 6 permutations with two ascents. The Eulerian numbers of the second kind, denoted by $\left\langle\!\!\left\langle {n \atop m} \right\rangle\!\!\right\rangle$, for a multiset with $n = 3$ different elements, thus, are: $\left\langle\!\!\left\langle {3 \atop 0} \right\rangle\!\!\right\rangle = 1$, $\left\langle\!\!\left\langle {3 \atop 1} \right\rangle\!\!\right\rangle = 8$ and $\left\langle\!\!\left\langle {3 \atop 2} \right\rangle\!\!\right\rangle = 6$. Here are some values arranged in the last triangle you will see for some time:

1	1						
2	1	2					
3	1	8	6				
4	1	22	58	24			
5	1	52	328	444	120		
6	1	114	1452	4400	3708	720	
7	1	240	5610	32120	58140	33984	5040

Again, this triangle shows some interesting properties. The sum of each row n is of course the doublefactorial of $2n - 1$, *e.g.* $\left\langle\!\!\left\langle {3 \atop 0} \right\rangle\!\!\right\rangle + \left\langle\!\!\left\langle {3 \atop 1} \right\rangle\!\!\right\rangle + \left\langle\!\!\left\langle {3 \atop 2} \right\rangle\!\!\right\rangle = 5!! = 15$ and $\left\langle\!\!\left\langle {4 \atop 0} \right\rangle\!\!\right\rangle + \left\langle\!\!\left\langle {4 \atop 1} \right\rangle\!\!\right\rangle +$

$\left\langle\!\left\langle {4 \atop 2} \right\rangle\!\right\rangle + \left\langle\!\left\langle {4 \atop 3} \right\rangle\!\right\rangle = 7!! = 105$. Neatly, the oughter right-hand diagonal shows the factorials. The last value of each row, $\left\langle\!\left\langle {n \atop n-1} \right\rangle\!\right\rangle$, hence, is $n!$.

Here comes the formula to compute the Eulerian numbers of the second kind recursively:

$$\left\langle\!\!\left\langle {n \atop m} \right\rangle\!\!\right\rangle = (2n - m - 1)\left\langle\!\!\left\langle {n-1 \atop m-1} \right\rangle\!\!\right\rangle + (m+1)\left\langle\!\!\left\langle {n-1 \atop m} \right\rangle\!\!\right\rangle, \tag{4.53}$$

with $\left\langle\!\left\langle {0 \atop 0} \right\rangle\!\right\rangle = 1$ and $\left\langle\!\left\langle {0 \atop m} \right\rangle\!\right\rangle = 0$. In Haskell this is:

```
eulerian2 :: Natural → Natural → Natural
eulerian2 0 0   = 1
eulerian2 0 _   = 0
eulerian2 n m = (2 ∗ n − m − 1) ∗ eulerian2 (n − 1) (m − 1)
              + (m + 1)         ∗ eulerian2 (n − 1) m
```

The Eulerian numbers, as many other things in mathematics, are named after Leonhard Euler (1707 – 1783), a Swiss mathematician and one of the most important mathematicians of all time. He is certainly the most important mathematician of his own time, and, for sure, the most productive one ever. He turned his family into a kind of math factory that produced thousands of papers in number theory, analysis, mechanics, optics, astronomy, ballistics and even music theory. He is also regarded as the founder of graph theory and topology. Modern terminology and notation is strongly based on Euler. He proved many theorems and made even more conjectures. But he was also a great math teacher, as his *Letters to a German Princess* show in which he lectured on mathematical subjects to non-mathematicians.

5. Primes

5.1. Gaps

The natural numbers are very simple in the sense that there are very simple ways to enumerate them all (given that we have infinite time and patience to do so, of course). Given a starting point, such as 0, 1 or any other number, we can generate all numbers from this point on just by counting up. This fact is so obvious that it appears ridiculous to visualise it like this:

1	2	3	4	5	6	7	8	9	10	...
1	2	3	4	5	6	7	8	9	10	...

A formula to generate this sequence might be: $f(n) = f(n-1) + 1$, *i.e.* the value for n equals the value of $n-1$ plus 1, which can be simplified to the closed form n. In fact, the function $f(n-1)+1$, starting with $n = 0$ is just the definition of natural numbers. The number used for counting, 1, is therefore called *unity*.

A slightly more interesting sequence is $f(n) = f(n-1) + 2$, that is is the multiplication table of 2, *i.e.* $f(n) = 2 \times n$:

1	2	3	4	5	6	7	8	9	10	...
2	4	6	8	10	12	14	16	18	20	...

This table shows exactly half of all numbers, *viz.* the even numbers. So, let us look at the second half of the numbers, the odd ones. The following table, correspondingly, shows a third of all numbers, namely those, divisible by 3:

1	2	3	4	5	6	7	8	9	10	...
3	6	9	12	15	18	21	24	27	30	...

Every second number in this table is even and, hence, already appears in the previous table. The logically next table, the one containing multiples of 4, would contain a quarter of all numbers. But it is not very interesting, since all numbers in that table already appear in the multiplication table for 2. With 5, however, we could get some new numbers to fill up the second half:

1	2	3	4	5	6	7	8	9	10	...
5	10	15	20	25	30	35	40	45	50	...

We see that every second number was already in the multiplication table for 2 and every third number in that for 3. So, it seems it is not too easy to get all numbers together – there is a lot of repetition in multiplication tables!

We can safely jump over the logically next table, 6, because all numbers in that table are already in the second table and, since 6 is a multiple of 3, in the third table as well. In the hope to find some more numbers of the second half, we continue with 7:

1	2	3	4	5	6	7	8	9	10	...
7	14	21	28	35	42	49	56	63	70	...

Every second number is in the table for 2, every third number appears also in the third table and every fifth number appears in the table for 5. The first new number we see is $7 \times 7 = 49$. It appears that we are running short of novelties! Indeed, we now have to skip 8 (since it is even), 9 (since it is a multiple of 3) and 10 (since it is not only even, but also a multiple of 5). The first number with some potential to bring something new is 11:

1	2	3	4	5	6	7	8	9	10	...
11	22	33	44	55	66	77	88	99	110	...

With some disappointment, we have to admit that there is no number up to $n = 10$ that we have not seen so far (besides 11 itself). With 12 we will not have more luck, since 12 is even. So let us have a look at 13 before we give up:

1	2	3	4	5	6	7	8	9	10	...
13	26	39	52	65	78	91	104	117	130	...

So, filling up the second half of the numbers does not appear to be an easy task. Very few numbers are really “new” in the sense that they are not multiples of numbers we have already seen. On second thought, this fact is not so curious anymore. The numbers that appear in a table for k have the form $n \times k$ and, for all $n < k$, we, of course, have seen n already in the tables for n, since $n \times k$ is the same as $k \times n$. In the table for 7, 7×7 was therefore the first number we had not yet seen. For numbers greater than k, the same is true for all multiples of earlier numbers; so 8×7, for instance, appears in the multiplication table of 4, since 8 is a multiple of 4. 9×7 appears in the table for 3, since 9 is a multiple of 3 and so on.

The really curious fact in this light is another one: that there, at all, are numbers that do not appear in tables seen so far. In fact, the numbers 2, 3, 5, 7, 11 and 13 never appear in any table, but their own. For 2 and 3, this is obvious, because 2 is the number we are starting with, so there simply is no table of a smaller number in which 2 could appear. Since 3, 5, 7 and so on are all odd, they cannot appear in the table for 2. But could they not appear in a later table? 5, obviously, cannot appear in the table for 3, since 5 is not a multiple of 3. The same holds for 7 and 7 is not a multiple of 5 either, so it will not appear in that table. We can go on this way with 11, 13 and any other number not seen so far and will always conclude that it cannot have appeared in an earlier table, since it is not a multiple of any number we have looked at until now.

We may think that this is a curiosity of small numbers until, say, 10 or so. But, when we go further, we always find another one: 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47. These unreachable numbers are called *primes* as opposed to *composites*, which are composed of other numbers in terms of multiplication. A prime, by contrast, is a number that is divisible only by 1 and itself.

Until now, we did not make a great effort to list prime numbers. In fact, almost every second number so far was prime. But it is hard to predict the next prime for any given n. For instance, what is the next prime after 47? Since, $48 = 2 \times 2 \times 2 \times 2 \times 3$, $49 = 7 \times 7$, $50 = 2 \times 5 \times 5$, $51 = 3 \times 17$ and $52 = 2 \times 2 \times 13$ none of these numbers is prime. The next one is only 53. So, what is the next prime after 6053 (which itself is prime)? Is there any more prime at all after this one or any other prime greater than the last prime we found so far in general? Let us examine how to find primes and which number is the last prime.

5.2. Finding Primes

In the previous section, we have already adopted a classical method to find primes, namely a *sieve*. We started by picking 2 and generated all multiples of 2. We then took the first number greater than 2 that was not in the list of multiples of 2, 3, and generated all its multiples. We continued by picking the first number greater than 3 that was neither in the list of multiples of 2 nor in that of multiples of 3, *viz.* 5. As a result, we identify primes as the heads of these list, $2, 3, 5, 7, 11, 13, \dots$.

This method was invented by Eratosthenes, a polymath of the 3^{rd} century BC, who was librarian of the library of Alexandria, which we already visited discussing Euclid. Eratosthenes wrote books on history, poetry, music, sports, math and other topics; he is considered founder of geography and he estimated the circumference of the earth remarkably close to the accurate value of about 40 000km known today. The exact precision of his estimate is subject to dispute – scholars refer to values he may have given between 39 500km, which would correspond to an error of only 1.5%, and 46 500km, which would be a more likely deviation of about 15%.

The *Sieve of Eratosthenes* goes as follows: write all numbers from 2 up to n, the upper limit of the range, for which you want to calculate the prime numbers. Eliminate all multiples of 2. Take the first number greater than 2 that was not yet eliminated and eliminate all its multiples. Take the first number greater than that number and proceed as before until the next number is n.

The following code shows an implementation in Haskell without upper limit, *i.e.* it finds all prime numbers exploiting lazy evaluation:

```
erato :: [Natural]
erato = sieve 2 [2..]
```

```
  where sieve x xs = case filter (λp → p `rem` x ≢ 0) xs of
                         [] → [x]
                         ps → x : sieve (head ps) ps
```

The function *sieve* has two arguments: the current number (starting with 2) and the list of all numbers starting with 2. The function filters all numbers out that leave remainder 0 divided by 2. If we have reached the last number, *i.e.* there are no more numbers in the list of all numbers, we just return the current number (which to know, if this case ever manifests, would be of the utmost interest). Otherwise, we insert the current number as head of the recursion on *sieve* that takes the head of the filtered list as the current number and the filtered list itself. In the first round we would get:

$\mathit{sieve}\ 2\ [2,3,4,5,6,7,8,9,...] = 2 : \mathit{sieve}\ 3\ [3,5,7,9,...]$

and continue with:

$\mathit{sieve}\ 3\ [3,5,7,9,...] = 3 : \mathit{sieve}\ 5\ [5,7,...]$
$\mathit{sieve}\ 5\ [7,...] = 5 : \mathit{sieve}\ 7\ [...]$

and so on.

If you call *erato* just like that, the function goes on forever, that is until the resources of your computer are exhausted or you interrupt the program. A call like *take n erato* would show the first n primes; a call like *takeWhile* ($<n$) *erato* would show all primes less than n; be careful: a call like *takeWhile* ($\not\equiv n$) *erato*, where n is not itself a prime, will run forever!

The principal merit of *erato* is its simplicity and conciseness; it is not very efficient however. Much more efficient is a very nice variation of the sieve of Eratosthenes given in the *Haskell Road.* This implementation is based on a primality test to decide whether a number enters the list of primes or not. The test itself uses a list of primes:

```
nextPrime :: [Natural] → Natural → Natural
nextPrime [] n       = n
nextPrime (p : ps) n | rem n p ≡ 0 = p
                     | p ↑ 2 > n   = n
                     | otherwise   = nextPrime ps n
```

nextPrime receives a list of primes and a number, n, to be tested for primality; if n is a prime, this number is returned, otherwise, the first prime dividing that number is returned.

If the first prime in the list, p, divides n, then p is returned; otherwise, if n is less than p^2, n is returned. Since p does not divide n and n is smaller than p^2 and p is the smallest prime remaining in the list, there will be no two primes in the list that multiplied with each other yield n. Because the primes remaining in the list are all greater than p, any product of two of them will obviously be greater than p^2 and, hence, greater than n. Therefore, n must be a prime. Otherwise, if n is greater than p^2, we continue the search with the next prime in the list.

The following code sequence turns *nextPrime* into a Boolean primality test:

```
prime :: Natural → Bool
prime n | n ≡ 1       = False
        | otherwise = ldp n ≡ n
ldp :: Natural → Natural
ldp = nextPrime allprimes
```

ldp stands for *least dividing prime* and yields the first prime number that divides n. It calls *nextPrime* with a list of all primes. The function is used in the test function *prime*, which compares n with *ldp* n, *i.e.* if the first prime that divides n is n, then n is a prime itself.

Now, where does the list of all primes come from? This is the beautiful part of the code. It is created in terms of *prime* used as a filter on the natural numbers:

```
allprimes :: [Natural]
allprimes = 2 : filter prime [3 ..]
```

Note that it is essential here to add 2 explicitly to the result, since it is 2 that bootstraps the algorithm. If we created *allprimes* as *filter prime* [2 . .], we would introduce an infinite regress: *allprimes* would try to filter prime numbers using *prime*, which, in its turn, uses *nextPrime* with *allprimes*, which, again calls *prime* to get a prime out of the list of numbers. With 2 already in that list, *prime* will first test primality of n, which is 3 in the first round, with the head of *allprimes*, *i.e.* 2. Since 2 does not divide 3 and $2^2 > 3$, 3 is returned and the algorithm is up and running.

A lot of sieves have been developed since Eratosthenes and many of them are much more efficient than Eratosthenes' sieve. A particular interesting one is the *Sieve of Sundaram*, which was developed by an Indian math student in the 1930ies. Sundaram's sieve finds the odd primes up to a limit of $2n + 2$ with the minor drawback that 2 is not in the list. Since 2 is the only even prime, this issue is easily solved by just adding 2 explicitly.

The algorithm is based on the fact that odd composites have odd factors. As we have already seen in the previous chapter, odd numbers can be represented as $2n + 1$. Odd composites, therefore, have factors of the form $(2i+1)(2j+1)$. If we multiply this out, we obtain $4ij+2i+2j+1$. We can split this sum into two terms of the form $(4ij+2i+2j)+1$ using the associative law. We move 2 out of the first term yielding $2(2ij + i + j) + 1$. Sundaram's algorithm cleverly removes all numbers of the form $2ij + i + j$ from the list of all numbers up to a given limit, doubles the remaining numbers and adds 1 to the result. Since all resulting numbers are again of the form $2n + 1$, they are all odd and, since all numbers of the form $2ij + i + j$ have been removed, we know that none of the resulting odd numbers $2n + 1$ is composite.

Here is a possible implementation:

```
sund :: Natural → [Natural]
sund n = 2 : [2 ∗ x + 1 | x ← [1 .. n] \\
                                [i + j + 2 ∗ i ∗ j | i ← [1 .. lim 0],
                                                     j ← [i .. lim i]]]
  where lim 0 = floor $ sqrt (fromIntegral n / 2)
        lim i = floor $ fromIntegral (n − i) / fromIntegral (2 ∗ i + 1)
```

We create the list of all numbers $[1 \mathinner{..} n]$ and subtract from it the list of all numbers of the form $i + j + 2ij$. the numbers i and j are generated as $[1 \mathinner{..} \mathit{lim}\ 0]$ and $[i \mathinner{..} \mathit{lim}\ i]$ respectively.

With the limits *lim*, we avoid multiplying pairs of numbers twice, such as 2 × 3 and 3 × 2. To achieve this, we generate i starting from 1 up to the greatest whole number less than the square root of half of n. The reasoning for this limit is that we do not want to generate too many is beyond n, since, at the end, we want to have primes only up to $2n + 2$, *i.e.* $i + j + 2ij <= n$ (since, at the end, we still multiply by 2). The smallest j we will use is i, which, injected into the above formula, yields $2i^2 + 2i$. For huge is, the second part is negligible, so instead of using this formula, we simplify it to $2i^2$ and generate is from 1 to a number x that squared and duplicated is at most eqal to n. This number, obviously, is the square root of half of n.

For j, the lower limit is i; the upper limit, $i + j + 2ij <= n$, can be expressed in terms of n and i: First we bring i on the other side: $j + 2ij <= n - i$; now we factor j out: $j(1 + 2i) <= n - i$, divide by $1 + 2i$: $j <= \frac{n-i}{2i+1}$ and get the limit defined in the code as *lim i*.

A bit confusing might be that we use *lim* 0 to calculate the limit for i. This is just a trick to use the same function for i and j. In fact, the limit for i is a constant relative to n. We could have it defined without an argument at all. But this way, using the same function for i and j, it looks nicer.

Sieves do a great job in creating lists of prime numbers. They are weak, when it comes to finding new prime numbers. Since sieves depend on primes discovered so far, any search for new prime numbers must start at the beginning, *i.e.* with 2. How far we will get, depends on available time and computing power. Those are serious limits in finding new primes, which may lie far ahead on the number ray. Instead of going forward step by step, as sieves do, we might want to make huge leaps forward ignoring thousands and millions of numbers.

A very simple method to guess a prime is based on the observation that primes often come in pairs. This, obviously, does not introduce huge leaps leaving thousands and millions of numbers out, it just leaves one number out. For instance 3 is prime and $3 + 2 = 5$ is prime too. Now, 5 is prime and $5 + 2 = 7$ is prime as well. 11 is prime and $11 + 2 = 13$ is prime too. So are 17 and 19, 29 and 31, 41 and 43, 59 and 61 and 71 and 73. But there are also many primes without a twin, *e.g.* 37, 53, 67, 83 and 89. In fact, if all primes came as twins, every second number would be prime and that, definitely, is not true. How many prime pairs there are and whether there are infinitely many of them is not known today. It is an unresolved problem in mathematics.

Bigger leaps are introduced by so called *Mersenne primes*, which have the form $2^n - 1$. This method of finding primes is based on the fact that many primes are powers of 2 minus 1, for instance: $3 = 2^2 - 1, 7 = 2^3 - 1, 31 = 2^5 - 1, 127 = 2^7 - 1$. Not all powers of 2, however, lead to Mersenne primes. $2^4 - 1 = 15$ is the composite number 3×5. $2^6 - 1 = 63$ is composite as well. So are $2^8 - 1 = 255 = 3 \times 5 \times 17, 2^9 - 1 = 511 = 7 \times 73, 2^{10} - 1 = 1023 = 3 \times 11 \times 31$. It turns out that all numbers of the form $2^n - 1$, where n is composite, are composite numbers as well. Mersenne primes are hence restricted to $2^p - 1$ with p a prime number. But even under this condition, not all Mersenne numbers are indeed primes, for instance $2^{11} - 1 = 2047 = 23 \times 89$. It is not known how many Mersenne primes there are and if there are infinitely many of them. This, again, is still an open problem in mathematics.

Most huge primes found today are Mersenne Primes. The search is assisted by the *Great Internet Mersenne Prime* Search (GIMP), which already found more than a dozen primes, most of them are also the greatest prime numbers known so far. The greatest of them is more than 10 million digits long.

Marin Mersenne (1588 – 1648), after whom Mersenne primes are named, was a priest who taught theology and philosophy in France. He wrote on philosopy and theology, but also on math and acoustics and he edited works of acient mathematicians such as Euclid and Archimedes. Mersenne was also a great organiser of science who corresponded with many mathematicians and scientists of his time including René Descartes, Galileo, Pierre Fermat and Etienne Pascal, the father of Blaise.

The main contribution to math is a list of Mersenne primes he compiled up to the exponent 257. There are some flaws in the list, he missed some primes and added some composite numbers. However, the effort is still impressive considering that all the math was done by hand.

Even greater leaps are introduced by searching for *Fermat primes*, named after our friend Pierre Fermat. These primes have the form $2^{2^n} + 1$. 3, for example, is a Fermat prime, since $2^{2^0} + 1 = 2^1 + 1 = 2 + 1 = 1$. 5, as well is a Fermat prime, since $2^{2^1} + 1 = 2^2 + 1 = 4 + 1 = 5$. The next Fermat prime is $2^{2^2} + 1 = 2^4 + 1 = 16 + 1 = 17$. The next is $2^{2^3} + 1 = 2^8 + 1 = 256 + 1 = 257$. The next, as you may have already guessed, is $2^{2^4} + 1 = 2^{16} + 1 = 65536 + 1 = 65537$. $2^{2^5} + 1 = 4294967297$, however, is not a prime, since $4294967297 = 641 \times 6700417$. The next one $2^{2^6} + 1 = 18446744073709551617$, as well, is composite, since $18446744073709551617 = 274177 \times 67280421310721$. As with Mersenne primes, we see that not all numbers constructed following the Fermat prime formula are actually primes. There is just a certain probability (which, hopefully, is greater than that of randomly picking a number) that a Fermat prime is indeed a prime. The largest Fermat prime known today is actually $2^{16} + 1$ and there is evidence that the number of Fermat primes is finite.

5.3. The fundamental Theorem of Arithmetic

The theorem with the somewhat bombastic name *fundamental theorem of arithmetic* states that every natural number other than unity either is a prime or can be expressed as a unique product of primes, its *prime factorisation.* "Unique", here, means that, for every natural number, there is exactly one prime factorisation.

On the first sight, this might appear trivial, but consider an example: 24 can be expressed as product in different ways, for instance: $24 = 3 \times 8$ and $24 = 4 \times 6$. 4, 6 and 8, however, are not prime numbers: $4 = 2 \times 2$, $6 = 2 \times 3$ and $8 = 2 \times 2 \times 2$ and, in consequence, the different products reduce to $24 = 2 \times 2 \times 2 \times 3$. In other words, there are many ways to obtain a number by multiplying other numbers, but there is only one way to obtain this number by multiplying prime numbers.

Let us have a look at some prime factorisations: 2 is prime and its prime factorisation is just 2. 3 is prime and its prime factorisation is just 3. 4 is composite and its prime factorisation is $2 \times 2 = 2^2$. Here are the numbers 5 to 20 and their prime factorisation:

$5 = \{5\}$
$6 = \{2, 3\}$
$7 = \{7\}$
$8 = \{2^3\}$
$9 = \{3^2\}$
$10 = \{2, 5\}$
$11 = \{11\}$
$12 = \{2^2, 3\}$
$13 = \{13\}$
$14 = \{2, 7\}$
$15 = \{3, 5\}$
$16 = \{2^4\}$
$17 = \{17\}$
$18 = \{2, 3^2\}$
$19 = \{19\}$
$20 = \{2^2, 5\}$

The facts constituting the fundamental theorem are known since antiquity and are outlined and proven in Book VII of the Elements. Many other proofs have been suggested since approaching the theorem from different angles. Indeed, its name already suggests that it is not an ordinary theorem, but is right in the centre of a whole bunch of problems in arithmetic.

In this section, we will look at one proof. We will basically establish that 1. every number can be factored into primes and 2. for any number, this factorisation is unique. We will also see, as a corollary to 1, that there are infinintely many primes.

The first theorem – that every number can be factored into primes – is quite simple. We distinguish two cases: the number is either prime or composite. If it is prime, the factorisation is simply that number. Otherwise, if it is not prime, there are two numbers, a and b such that $a \times b = n$, where n is the number in question. Now, a and b either are prime numbers or there are other numbers a_1 and a_2, such that $a_1 \times a_2 = a$, and b_1 and b_2, such that $b_1 \times b_2 = b$. Indeed, that a number is composite means essentially that there exist two other numbers that divide this number. Therefore, every number can be expressed as a product of prime numbers. $\square$

Now we will prove the fundamental theorem of arithmetic. We will use a proof technique called *indirect proof* or *proof by contradiction.* We have already used this type of proof silently and it is in fact a quite common tool in reasoning. The strengths of indirect proofs are that they are often very simple, much simpler than direct proofs, and that they can prove things that we cannot demonstrate. The latter is of major importance, in particular when talking about infinitely big or small things. We can prove, for example, that there are infinitely many primes without the need to construct infinitely many primes.

Indirect proofs also have some drawbacks, though. The most important one is that they do not provide a method to compute the result. An indirect proof may be used to prove the existence of something, but does not provide a method to construct that "something" (such as the result of a given function or the greatest prime number). They are therefore sterile in the sense that we obtain only the abstract knowledge that some theorem is true, but no further insight into the concepts under investigation and no new methods to work with these concepts. That is quite poor and, indeed, many mathematicians have expressed their inconvenience or even disgust when confronted with indirect proofs. There is even a philosophical tradition within mathematics, *mathematical constructivism*, that aims to find direct proofs for all mathematical theorems for which only indirect proofs are known today. Without taking side in the philosophical debate, most mathematicians would agree today that an indirect proof should be the last resort, *i.e.*, when there is a direct proof, it should be preferred.

So, what is an indirect proof in the first place? The proof of a theorem A works by demonstrating that the assumption $\neg A$ leads to a contradiction. We therefore start the proof by stating what we assume to be true. Let us look, as a simple example, a variant of Euclid's proof that there are infinitely many primes:

Assume that there is a finite number of primes. Then we can enumerate the set P of all primes as $P = \{2, 3, 5, \dots, p\}$, where p is the last prime. The product of the primes in this set is a composite number: $n = 2 \times 3 \times 5 \times \dots \times p$. So, what about $n + 1$?

This number is either prime, then P was incomplete, which immediately contradicts our assumption; or it is composite and then it has a prime factorisation. But none of the primes in P can be part of that factorisation, because no number greater than 1 divides both n and $n+1$: 2 divides n and $n+2$, but not $n+1$; 3 divides n and $n+3$, but not $n+1$, p divides n and $n+p$, but not $n+1$. Therefore, there must be at least one prime that is not in P, which, again, contradicts our assumption. □

There are, hence, infinitely many primes.

We now prove the fundamental theorem of arithmetic, *i.e.* that there is only one way to factor any given number into primes. We prove this by contradiction and assume that there is at least one number for which it is actually possible to find more than one prime factorisation. We must be very cautious about such assumptions, since we want the contradiction to hit the right place. We, therefore, assume that there is *at least* one number without assuming anything further – there may be just that one number, there may be many or it may be even true for all composites that there is more than one prime factorisation.

In the following, however, we will talk about just one such number. If there is only one number with that property, we talk about that one. Otherwise, if there are many, we talk about the smallest number with that property. We can simply verify for small numbers, in particular 4, the smallest composite number, that there is only one prime factorisation, *i.e.* 2×2. If there is a number with more than one factorisation, it is definitely greater than 4.

We call this smallest number for which more than one prime factorisation exist m:

$$m = p_1 \times p_2 \times \cdots \times p_r \tag{5.1}$$

$$m = q_1 \times q_2 \times \cdots \times q_s \tag{5.2}$$

The ps and qs in these equations are all primes. Also, the ps and qs differ, such that at least one p is not in the list of qs and vice versa. To illustrate this, the ps could be 3 and 8 (if 8 was a prime number) and the qs could be 4 and 6 (if 4 and 6 were prime numbers) in the factorisations of 24. 4, 6 and 8, of course, are not prime numbers. But what we claim (to, hopefully, create a contradiction) is that there are numbers for which different decompositions are possible even with prime numbers.

We further assume that the two factorisations, the ps and qs above are ordered, such that

$$p_1 \leq p_2 \leq \cdots \leq p_r$$

and

$$q_1 \leq q_2 \leq \cdots \leq q_s.$$

Now, p_1 and q_1 must be different, *i.e.* either $p_1 < q_1$ or $q_1 < p_1$, for, if $p_1 = q_1$, we could divide both sides, the p-factorisation and the q-factorisation, by p_1 and obtain a number with two factorisations that is actually smaller than m – but we assume that m is the smallest number with that property. This assumption forces us to also assume that either $p_1 < q_1$ or $q_1 < p_1$. Let us say that p_1 is the smaller one. (It is irrelevant which one it actually is. If we chose q_1 to be the smaller one, we would just swap ps and qs in the following equations.) We can now compute a number m':

$$m' = m - (p_1 \times q_2 \times \cdots \times q_s), \tag{5.3}$$

for which it, obviously, holds that $0 < m' < m$, *i.e.* m' is smaller than m and, hence, has a unique prime factorisation (since m is the smallest number with the property that it has more than one prime factorisation).

Now, by substituting for m, we derive:

$$m' = (p_1 \times p_2 \times \cdots \times p_r) - (p_1 \times q_2 \times \cdots \times q_s) \tag{5.4}$$

and, by factoring p_1 out, we get:

$$m' = p_1 \times (p_2 \times p_3 \times \cdots \times p_r - q_2 \times q_3 \times \cdots \times q_s) \tag{5.5}$$

and clearly see that p_1 is a factor of m'. But we can also derive

$$m' = (q_1 \times q_2 \times \cdots \times q_s) - (p_1 \times q_2 \times \cdots \times q_s), \tag{5.6}$$

from which, by dividing by $q_2 \times q_3 \times \cdots \times q_s$, we can further derive

$$m' = (q_1 - p_1) \times (q_2 \times q_3 \times \cdots \times q_s). \tag{5.7}$$

Since p_1 is a factor of m', it must be a factor of either $q_1 - p_1$ or $q_2 \times q_3 \times \cdots \times q_s$. (Remember that there is only one way to factor m' into primes, since it is smaller than m, the smallest number with more than one prime factorisation.) It cannot be a factor of $q_2 \times q_3 \times \cdots \times q_s$, since all the qs are primes and greater than p_1. So, it must be a factor of $q_1 - p_1$. In other words, there must be a number, say, h for which it holds that

$$q_1 - p_1 = p_1 \times h \tag{5.8}$$

By adding p_1 to both sides we get $q_1 = p_1 \times h + p_1$. By factoring p_1 out on the right-hand side of the equation we obtain:

$$q_1 = p_1 \times (h + 1) \tag{5.9}$$

In other words, p_1 is a factor of q_1. But this is a contradiction, since q_1 is prime. □

5.4. Factoring

In the previous section, we have made heavy use of factoring, *i.e.* of decomposing a number into its prime factors. But we did so without indicating an algorithm. We have just stated that $9 = 3 \times 3$ or $21 = 3 \times 7$. For such small numbers, that is certainly acceptable – we have learnt our multiplication tables in school and can immediately say that $32 = 2^5$. With bigger numbers, this becomes increasingly difficult and, therefore, we clearly need an algorithm.

Unfortunately, no efficient factorisation algorithm is known. In fact, there is one that is sufficiently fast, but that one does not run on traditional computers. It is a quantum algorithm. It was developed by the American mathematician Peter Shor in 1994. It has already been implemented on real quantum computers several times and in 2012 it was used to factor 21. That is not the greatest number factored by a quantum computer so far. With another approach, not involving Shor's algorithm, but a simulation method called *abbiatic quantum computation*, a Chinese team achieved to factor 143 as well in 2012.

We will not enter the quantum world here, but rather stick to classical algorithms, even if this leaves us with highly inefficient programs that need exponential time to factor numbers. There is something good about the fact that factorisation is hard: many algorithms in cryptography are based on it. So, would factorisation be made much simpler overnight, our online banking passwords and other data would not be secure anymore. On the other hand, there is no proof that factorisation will remain a hard problem forever. It is not even known to which *complexity class* factorisation belongs.

Complexity classes, in theoretical computer science, descibe the level of difficulty of solving a problem. There are a lot of complexity classes; for the moment, two are sufficient: P and NP. P stands for *polynomial time.* Polynomial refers to formulas of the form $n^a + c$ or similar, where n is the size of the input (for example the number we want to factor) and a and c are constants or even other – but similar – formulas. Essential is that, in formulas that describe the cost of algorithms that are solutions to P-problems, n does never appear as exponent. Algorithms, whose cost is described by formulas where n appears in exponents, *e.g.* a^n, are exponential. Such algorithm are considered unfeasible for input of relevant size.

The interesting point about NP, now, is that solutions for problems in this class need an incredible amount of steps, such as exponential time where n appears in the exponent, but, if a potential answer is known, it is extremely easy to verify if this answer is correct. The acronym NP means *non-deterministic polynomial time*. The name refers to the fact that if an answer is known it can be verified in polynomial time (the verification, hence, is a P-problem). Where the answer comes from, however, is unclear – it appears out of the blue, in a non-deterministc way. It could be chosen randomly, for instance, or a magus, like Merlin, could have suggested it. Notice that this is definitely a characteristic of factorisation. Given a number such as 1771, it may be hard to say what its prime factors are. If we were told, however, that 7 is one of the factors, we can use division to verify that $1771 \bmod 7 = 0$ and even to reduce the problem to finding the factors of $1771/7 = 253$.

Today, factorisation is not considered to be in NP. Shor's algorithm is a quantum probabilistic algorithm (belonging to class BQP – *bounded-error quantum probabilistic*) and, therefore, it is assumed that it may belong also to a classic probabilistic class (such as BPP – *bounded-error probabilistic polynomial*). This assumption is supported by the fact that there appears to be a consistent distribution of primes – but more on that later.

Factorisation is an area with extensive research. To that effect, there are many algorithms available, most of them exploiting probabilistic in some way or another. We will stick to an extremely simple approach that, basically, uses a trial-and-error method. We simply go through all prime numbers, for this purpose we can use one of the prime number sieves, and try to divide the input number by each one until we find a prime that divides this number. If we do not find such a prime, the number must be prime and its factorisation is just that number.

Here is the searching algorithm:

```
findf :: Natural → Natural → [Natural] → (Natural, Natural)
findf n _ [ ] = (1, n)
findf n l (p : ps) | p > l        = (1, n)
                   | otherwise = case n ‘quotRem‘ p of
                                   (q, 0) → (p, q)
                                   (_, r) → findf n l ps
```

The function *findf* receives three arguments: the input number, n, an upper limit, l, and the list of primes. It yields a pair of numbers: the first is the prime number that divides n, the second is the quotient of n divided by the prime. If the list of primes is exhausted, a case that, as we know from the previous section, is extremely rare, we just return $(1, n)$ to indicate that we have not found a proper solution. Otherwise, we check if we have reached the upper limit. In this case, we again yield $(1, n)$ to signal that no proper solution was found. Otherwise, we divide n by the first prime in the list and, if the remainder is zero, we yield this prime and the quotient. Otherwise, we just continue with the remainder of the prime list.

The result of this function, hence, is one prime factor and another number that, multiplied by the factor is n. If this other number is prime as well, we are done. Otherwise, we must continue factoring this other number:

```
trialfact :: Natural → [Natural]
trialfact 0 = []
trialfact 1 = []
trialfact n = let l = fromIntegral $ floor $ sqrt (fromIntegral n)
              in case findf n l allprimes of
                   (1, _) → [n]
                   (p, q) → if prime q then [p, q]
                                        else p : trialfact q
```

This function receives the number to be factored and yields the list of factors of this number. 0 and 1 cannot be factored. The result in this case, hence, is simply the empty list. For all other cases, we first determine the upper limit as the greatest natural number less than the square root of n. We already discussed the reasoning for this upper limit in the context of the Sundaram sieve: we assume an ordered list of primes and, when the current prime is greater than this limit, the square root of n, the product of any two primes greater than this prime will necessarily be greater than n. There is hence no need to continue the search.

We then call *findf* with n, the limit l and *allprimes*. If the result is $(1, \_)$, *i.e.* if *findf* has not found a proper solution, then n must be prime and we return $[n]$ as the only factor. For other results, we know that p, the first of the pair, is a prime. We do not know this for the second of the pair, which may or may not be prime. If it is prime, we yield $[p, q]$. Otherwise, we call *trialfact* with q and add p to the result.

We could skip the primality test and just continue with *trialfact* q, since, if q is prime, *findf* will yield $(1, q)$ and then *trialfact* would yield $[q]$ anyway. In the hope of finding a primality test that is more efficient then the test we have defined so far (which, itself, uses a sieve to construct *allprimes*), we use this explicit test to obtain some speed-up in the future.

We can use *trialfact* to investigate the distribution of primes further. We start with the numbers $2 \dots 32$ and create the list of factorisations of these numbers calling *map trialfact* $[2\,..\,32]$:

```
[2]
[3]
[2,2]
[5]
[2,3]
[7]
[2,2,2]
[3,3]
[2,5]
[11]
[2,2,3]
[13]
[2,7]
[3,5]
[2,2,2,2]
[17]
[2,3,3]
[19]
[2,2,5]
[3,7]
[2,11]
[23]
[2,2,2,3]
[5,5]
[2,13]
[3,3,3]
[2,2,7]
[29]
[2,3,5]
[31]
[2,2,2,2,2]
```

What jumps immediately into the eyes is the fact that the lists appear to grow in length – with sporadic prime numbers to appear in between slowing down the growth. Here is the list of the sizes of the factorisations $2 \dots 32$:

1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 3, 1, 2, 2, 4, 1, 3, 1, 3, 2, 2, 1, 4, 2, 2, 3, 3, 1, 3, 1, 5.

Most factorisations $(2 \dots 32)$ are of size 1 or 2, 1 being the size of prime number factorisations, which consists only of that prime number. Greater numbers appear sporadically, 3, 4 and 5, and seem to grow – in-line with our previous observation. Those greater numbers are certainly caused by repetition of prime numbers, such as $2 \times 2 \times 2 \times 2 = 16$. How would it look if we counted only unique primes? Let us have a try:

1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 2, 1, 2, 1, 3, 1, 1.

The factorisations, now, grow much slower. The first factorisation with more than 3 distinct primes appears only with 30 (2 × 3 × 5). All other factorisations have either size 1 or 2. Factorisations of size 1 are those of the primes and those containing only repeated primes. But there are no clear patterns that would reveal some regularity among factorisations. Perhaps, it could be helpful to look at the distinction odd versus even sized factorisations? To do that, we should distinguish between factorisations with and without repeated primes. We could, for instance, say that factorisations with repeated primes have the value 0; odd-sized factorisations have value -1 and even-sized factorisations have value 1.

This rule describes the Möbius function, which we could define as:

```
moebius :: Natural → Integer
moebius = chk ∘ trialfact
  where chk f | f ≢ nub f        = 0
              | even (length f)  = 1
              | otherwise        = −1
```

The *moebius* function for a number n checks whether the factorisation of that number contains repeated primes ($f \not\equiv nub\ f$); if so, the result is 0. Otherwise, if the number of primes in the factorisation is even, the result is 1. Otherwise, the result is -1. Notice that we use *Integer* as output data type, since -1 is not a natural number.

Here are the values of the Möbius function for the numbers $2 \dots 32$:

-1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0.

It is still difficult to see regularities. What, if we defined an accumulated Möbius function where each value corresponds to the sum of the values of the Möbius function up to the the current number:

```
mertens :: Natural → Integer
mertens n = sum (map moebius [1 . . n])
```

Mapped on the numbers $2 \dots 32$, this function gives:

0, -1, -1, -2, -1, -2, -2, -2, -1, -2, -2, -3, -2, -1, -1, -2, -2, -3, -3, -2, -1, -2, -2, -2, -1, -1, -1, -2, -3, -4, -4.

This still does not reveal convincing patterns. Apparently, most numbers have a negative value, but this is true only for the small section we are looking at. The result for *mertens* 39 and *mertens* 40, for instance, is 0. The values for 94 to 100 are all positive peaking with 2 at 95 and 96.

This investigation appears to remain fruitless and we should give it up at least for the moment. We will come back to Möbius and Mertens, however. Even if not visible on the surface, there is something about the concept.

The Möbius function was invented by August Ferdinand Möbius (1790 – 1868), a German mathematician and astronomer, student of Gauss in Göttingen. There are many unusual concepts discovered or developed by this man, for instance, the famous *Möbius strip*, a two-dimensional surface with the uncommon property of having only one side in three-dimensional space.

Franz Mertens (1840 – 1927), after whom the Mertens function is named, is less known. He proposed the Mertens function together with a conjecture concerning its growth that, if proven correct, could have been used to prove the *Riemann Hypothesis* on the distribution of primes. But, unfortunately, Meterns' conjecture was proven wrong and the Riemann Hypothesis remains an enigma until today.

5.5. Factoring Factorials

In the next sections, we will dive into group theory related to primes. There is a problem that makes a very nice link between factoring and the upcoming investigations: factoring factorials.

The factorial of a number n is defined as the product of all numbers $1 \dots n$: $1 \times 2 \times 3 \times \dots \times n$. The immediate consequence of this definition is that all primes in the range $1 \dots n$ are factors of $n!$ and that all factors of $n!$ are primes in the range $1 \dots n$. The first fact is easy to see: since we multiply all numbers $1 \dots n$ with each other, all primes in this range must be part of the product. Furthermore, all composites in this range are products of prime factors in this range and, hence, $n!$ is the product of products of the primes between 1 and n.

For the second fact to become clear, assume for a moment that there were a prime $p > n$ that is a prime factor of $n!$ That would mean that some product of primes $< n$ would result in that p. But that is impossible, since p is a prime. It is not a product of other primes and can therefore not result from multiplying other primes and can thus not be a prime factor of $n!$

To find the prime factors of $n!$ (and, hence, $n!$ itself), we have to ask how often each prime appears in the factorisation of $n!$? This leads to the question how many numbers in the range $1 \dots n$ are actually divisible by a given prime. This is easily answered, when we realise that the range $1 \dots n$ consists of n consecutive numbers. For any number a, every a^{th} number is divided by a. There are, hence, $\lfloor n/p \rfloor$ numbers that are divided by p. Let us look at the example $n = 6$ and $p = 2$. The product $n!$ consists of six numbers: $1, 2, 3, 4, 5, 6$. Every second number is even, namely 2, 4 and 6 itself. This is $6/2 = 3$ numbers. Therefore, 2 must appear at least 3 times as factor in $n!$

But wait: 2 appears 2 times in 4, since the factorisation of 4 is 2^2. How can we tell how many of the numbers in the range are divided by 2 more than once? Well, we just do the same, we divide 6 by 4, since every fourth number is divided by 4. Since $\lfloor 6/4 \rfloor = 1$, there is only one number in the range $1 \dots 6$ that is divided by 4 and, hence, divided twice by 2, *viz.* 4 itself. When we add the two results $\lfloor 6/2 \rfloor = 3$ and $\lfloor 6/4 \rfloor = 1$, we get 4. In other words, there are 3 numbers divided by 2 and 1 number divided by 2 twice. Therefore, 2 appears 4 times in the prime factorisation of 6! Let us check if this result is correct: $6! = 720$. The prime factorisation of 720 is *trialfact* 720: $[2,2,2,2,3,3,5]$. So the result is correct.

Let us try to confirm the result for the other primes ≤ 6, namely 3 and 5. $\lfloor 6/3 \rfloor$ is 2; there are hence two numbers divided by 3 and these numbers are 3 and 6. Since $3^2 = 9$ is already greater than 6, there is no number in the range of interest that is divided by 3 twice. Therefore, there are two occurrences of 3 in the prime factorisation of 6! $\lfloor 6/5 \rfloor$ is 1, which means there is only one number divided by 5, namely 5 itself. 5, therefore appears once in the factorisation of 6! With this approach, we arrive at the correct result: $6! = 2^4 \times 3^2 \times 5 = 720$.

We can implement this approach in Haskell to get a speed-up on the factorial computation compared to the laborious multiplication of all numbers $1 \dots n$. We first implement the logic to find the number of occurrences for one prime:

```
pInFac :: Natural → Natural → Natural
pInFac n p = p ↑ (go p 1)
  where go q e = let t = n `div` q
                 in if t ⩽ 1 then t
                              else let e′ = e + 1 in t + go (p ↑ e′) e′
```

In *go*, which is called with a prime number p and and expononent $e = 1$, we first compute the quotient $\lfloor n/q \rfloor$. If this quotient is 1 or 0, we immediately yield this number. Otherwise, we continue with p raised to $e + 1$. That is, if $n = 6$ and $p = 2$, then we first compute $t = 6$ `` `div` `` 3, which is 2 and hence greater than 1. We now increment e, which, initially is 1, and call *go* with $2^2 = 4$ and $e = 1 + 1 = 2$.

In the next round t is 6 `` `div` `` 4, which is 1. We return immediately and add 1 to the previous value of t, which was 3, obtaining 4. The function overall yields p raised to the result of *go*, hence $2^4 = 16$.

We call this function in the following code:

```
facfac :: Natural → [Natural]
facfac n = go allprimes
  where go (p : ps) = let x = pInFac n p
    in if x ≡ 1 then [] else x : go ps
```

The function *facfac* results in a list of *Natural*, *i.e.* it yields the factors of $n!$, sucht that *product* (*facfac* n) is $n!$ It calls the internal function *go* on *allprimes*. On the first prime, it calls *pInFac* n p. If the result is 1, *i.e.* if we raised p to zero, we terminate with the empty list. Otherwise, we continue with the tail of *allprimes*.

Here are the results for the numbers 2...12:

$[2]$
$[2, 3]$
$[8, 3]$
$[8, 3, 5]$
$[16, 9, 5]$
$[16, 9, 5, 7]$
$[128, 9, 5, 7]$
$[128, 81, 5, 7]$
$[256, 81, 25, 7]$
$[256, 81, 25, 7, 11]$
$[1024, 243, 25, 7, 11]$

It would be very interesting, of course, to know how much faster *facfac* is compared to the ordinary recursive *fact*. Well, *fact* multiplies n numbers with each other. There are hence $n - 1$ multiplications. By contrast, *facfac* calls *pInFac* for every prime less than or equal to n. *pInFac* is somewhat more complex in computation than multiplication, but when the difference between n and the number of primes up to n is significant, then the difference between the cost of multiplication and that of *pInFac* does not matter. The question remains: how many primes are there among the first n numbers?

5.6. Arithmetic modulo a Prime

On the first sight, there is nothing special about arithmetic modulo a prime. It is plain modular arithmetic where the number to which all the number operations are taken modulo happens to be a prime. However, as we will see in this section, something very significant changes, when it actually is a prime. Let us first recall the properties of modular arithmetic and define a Haskell data type to model it.

As we have seen before, numbers modulo a number n repeat cyclicly, that is any number modulo n is a number in the range $0 \dots n - 1$. When we think of the clock again, any number, independent of its size, reduces to a number $0 \dots 11$ taken modulo 12: 1 mod 12 is 1, 5 mod 12 is 5, 10 mod 12 is 10 and 13 mod 12 is again 1, 17 mod 12 is 5 and 22 mod 12 is just 10 again. In other words, the numbers modulo a number n form a finite groupoid (or magma) whith addition and multiplication, that is, addition and multiplication are closed under natural numbers modulo n:

$5 + 3 = 8 \bmod 12 = 8$
$5 + 10 = 15 \bmod 12 = 3$
$13 + 15 = 28 \bmod 12 = 4$
$15 + 17 = 32 \bmod 12 = 8$

and

$5 \times 3 = 15 \bmod 12 = 3$
$5 \times 10 = 50 \bmod 12 = 2$
$13 \times 15 = 195 \bmod 12 = 3$
$15 \times 17 = 255 \bmod 12 = 3.$

If we consider more complex sums and products of the form $a+b+\dots+c$ and $a\times b\times\dots\times c$, it becomes apparent that it is more efficient to take the terms and factors modulo n before applying the operation:

$13 \bmod 12 + 15 \bmod 12 = 1 + 3 = 4$
$15 \bmod 12 + 17 \bmod 12 = 3 + 5 = 8$

or:

$13 \bmod 12 \times 15 \bmod 12 = 1 \times 3 = 3$
$15 \bmod 12 \times 17 \bmod 12 = 3 \times 5 = 15 \bmod 12 = 3.$

A short note on terminology may be in order here. You may have realised that two different operators for the modulo are used: one is a binary operator, which we use to indicate that a number is taken modulo another number: $a \bmod n$. The other is an indicator that an operation is taken modulo: $a + b \mod n$ (with a bigger gap between b and "mod"). The point is to avoid confusion between $a + b \bmod n$, that would be a number a not modulo another number $+b$ modulo n. For example: $15 + 28 \bmod n = 19$, but $15 + 28 \mod n = 7$ where the whole operation is taken modulo n. Since the distinction between "mod" and " mod " is quite subtle, we will use the convention that, if not obvious from the context or indicated otherwise, we usually take operations modulo n.

Modular arithmetic looks a bit weired at the beginning, for instance $13+15 = 4$ definitely looks wrong. But, in fact, nothing special has changed. We see that the associativity law holds:

$13 + (15 + 17) \mod 12 = (13 + 15) + 17 \mod 12 = 9,$
$13 \times (15 \times 17) \mod 12 = (13 \times 15) \times 17 \mod 12 = 3.$

There is also an identity for each, addition and multiplication. For addition this is 0 (and, thus, all integer multiples of 12):

$0 \bmod 12 = 0$
$24 \bmod 12 = 0$
$36 \bmod 12 = 0$
$\dots$

For multiplication, the identity is 1 (and, hence, all multiples of 12 plus 1):

$1 \bmod 12 = 1$
$25 \bmod 12 = 1$
$37 \bmod 12 = 1$

...

It holds of course that for any a divisible by 12:

$$a + b \mod n = b$$

and

$$(a + 1) \times b \mod n = b.$$

Of course, also the distributive law holds:

$$a \times (b + c) \mod n = ab + ac \mod n.$$

This altogether means that numbers modulo n form a semiring with addition and multiplication just as the natural numbers. The difference between natural numbers and numbers modulo n is that the set of natural numbers is infinite whereas the set $1 \dots n-1$ is of course finite.

Let us have a look at how we can model such a modular semigroup with Haskell. We first define a data type *Module*:

```
data Module = Module Natural Natural
```

A *Module* according to this definition is created by two natural numbers. The first is the modulus n to which we take the second modulo. For instance, *Module* 12 13 is 13 mod 12. To enforce modular arithmetic from the beginning, we provide a constructor:

```
tomod :: Natural → Natural → Module
tomod n a = Module n (a ‘rem‘ n)
```

For all (binary) operations on *Module*s, we want to ensure that both parameters have the same modulus. Operating on two *Module*s with different moduli leads to wrong results. For this reason, we will use a guard on all operations:

```
withGuard :: (Module → Module → r) → Module → Module → r
withGuard o x@(Module n  a)
                    y@(Module n′ b) | n ≢ n′     = error "different moduli"
                                    | otherwise = x ‘o‘ y
```

Now, we make *Module* instance of some type classes:

```
instance Show Module where
  show (Module n a) = show a
instance Eq Module where
  (≡) = withGuard (λ(Module _ a) (Module _ b) → a ≡ b)
instance Ord Module where
  compare = withGuard (λ(Module _ a) (Module _ b) → compare a b)
```

We *show* the number actually taken modulo n and consider n known in the context. This is much more convenient to read, even if some information is lost.

To check for equality and to compare two *Module*s we apply the guard to the operations, $(\equiv)$ and *compare* respectively.

The next listing shows addition and multiplication:

```
add :: Module → Module → Module
add (Module n a) (Module _ b) = Module n ((a + b) `rem` n)
mul :: Module → Module → Module
mul (Module n a) (Module _ b) = Module n ((a ∗ b) `rem` n)
```

Since the numbers a and b are already modulo n, taking the results of the computations modulo n is an inexpensive operation. Since the maximum for both, a and b, is $n-1$, $a+b$ is at most $2n-2$ and, taking this modulo n, is just $2n-2-n=n-2$.

The greatest value the product $a \times b$ can achieve is $(n-1)(n-1) = n^2 - 2n + 1$. To reduce this value modulo n, one division step is needed and that is indeed the worst case for modular multiplication.

Now, what about subtraction? Subtracting two numbers modulo n should also be in the range $0 \dots n-1$, but what happens, when the second number is greater than the first one? The normal subtraction beyond zero gives $a - b = -(b - a)$. In modular arithmetic, a negative number $-k$ is interpreted as $n-k$, *i.e.* the minus sign is interpreted as counting down from n, which in fact is the same as counting down from 0, since $n \bmod n$ is just 0.

We can therefore, when we have a negative number in the range $-(n-1)\dots 0$, just add n to the result: $a - b = -(b-a) + n = n - (b - a)$. For instance for $n = 13$: $3 - 9 + 12 = -6 + 12 = 6$. Note that subtraction handled like this is the inverse of addition: $3 - 9 = 6 \mod 12$ and $6 + 9 = 3 \mod 12$. The point is that addition modulo n with two numbers already modulo n is at most $2n-2$. The remainder of any number up to $2n-2$ is just this number minus n: $6 + 9 = 15 - 12 = 3$. For subtraction, a similar is true: the smallest value, subtraction can produce is $n-1$ (in the case of $0-11$, for instance). The inverse of this operation for negative numbers is then just adding n instead of subtracting it.

We, hence, can implement subtraction as:

```
sub :: Module → Module → Module
sub (Module n a) (Module _ b) | a < b       = Module n (a + n − b)
                              | otherwise = Module n (a − b)
```

Note that we change the order of the operations for the case that $a < b$. If we performed $a - b$ first, we would subtract beyond zero. Even if the overall result is again a natural number, the intermediate result is not. Therefore, we first add a and n and then we subtract b. In spite of handling problems of natural numbers only, we are on the verge of entering new territory. But we can state a very exciting result: In modular arithmetic, natural numbers and addition form a group: addition is closed, addition adheres to the associativity law, there is an identity, namely 0 (and all multiples of the modulus n), and, for any number modulo n, there is an inverse element.

With addition, subtraction and multiplication defined, we can now make *Module* an instance of *Num*:

```
instance Num Module where
  (+)   = withGuard add
  (−)   = withGuard sub
  (∗)   = withGuard mul
  abs a = a
  signum (Module n a) = (Module n (signum a))
  fromInteger i = Module (fromInteger (i + 1)) (fromInteger i)
```

The basic arithmetic operations are defined as *add*, *sub* and *mul* with the guard to avoid arithmetic on different moduli. As with natural numbers, we ignore *abs*, since all natural numbers are positive. *signum* is just the *signum* of *a*, *i.e.* a *Module* with 0 for 0 and 1 for any number greater than 0.

fromInteger is a bit tricky. We cannot convert an integer to a *Module* "as such". To convert an integer, we must know to which n the number should be taken modulo. When we say that, by default, an integer i is $i \bmod (i+1)$, the value of that module is always i, for instance: $2 \bmod 3 = 2$. This appears to be a reasonable default value.

Division, as usual, is not so easy. We would like to define division in a way that it serves as inverse of multiplication, *e.g.* $a \times b/a = b$. That means that for any number a, we want a number a', such that $a \times a' = 1$. Consider the example $n = 6$ and $a = 3$:

$3 \times 0 = 0 \mod 6$
$3 \times 1 = 3 \mod 6$
$3 \times 2 = 0 \mod 6$
$3 \times 3 = 3 \mod 6$
$3 \times 4 = 0 \mod 6$
$3 \times 5 = 3 \mod 6.$

This looks strange: any multiplication of 3 modulo 6 creates either 0 or 3, but not 1,2,4 or 5. The point is that 3 divides 6, in particular $2 \times 3 = 6$. For this reason, 6 divides every second product of 3 greater than 3, *i.e.* $6, 12, 18, 24, \dots$ The other half of multiples are just those that leave a remainder of 3 divided by 6.

What, if a does not divide n, like for example with $a = 6$ and $n = 9$?

$$\begin{aligned}
6 \times 0 &= 0 \mod 9\\
6 \times 1 &= 6 \mod 9\\
6 \times 2 &= 3 \mod 9\\
6 \times 3 &= 0 \mod 9\\
6 \times 4 &= 6 \mod 9\\
6 \times 5 &= 3 \mod 9\\
6 \times 6 &= 0 \mod 9\\
6 \times 7 &= 6 \mod 9\\
6 \times 8 &= 3 \mod 9.
\end{aligned}$$

We see more variety, but, still, we have only 3 numbers out of 9 possible. Now, every third multiple of six is divisible by 9. This is, of course, because 3 divides 9 and also divides 6. In consequence every third multiple of 6 is divisible by 9. When we think this through, we see that there are indeed many numbers n divisible by numbers $1 \dots n - 1$. Only if n is *coprime* to those numbers, all of them would appear as result of multiplication of any two numbers $a, b \in 1 \dots n - 1$. That two numbers, a and b, are coprime means that they have no common factors, *i.e.*: $\gcd(a, b) = 1$. If we look at an example, where n is coprime of all numbers $1 \dots n - 1$, we see that all numbers $0 \dots n - 1$ actually appear as results of multiplications of two numbers in the range:

$$\begin{aligned}
3 \times 0 &= 0 \mod 5\\
3 \times 1 &= 3 \mod 5\\
3 \times 2 &= 1 \mod 5\\
3 \times 3 &= 4 \mod 5\\
3 \times 4 &= 2 \mod 5.
\end{aligned}$$

The point is that 5 has no common divisor with any of the numbers $0 \dots 4$. We know that for sure, because 5 is a prime number. In consquence, no multiple of any number a from the range $1 \dots 4$ will be divisible by 5 but those that are also multiples of 5. For this reason, any multiple of a number in the range will again leave a remainder in the range when divided by 5. The only exceptions are multiples of the form ka where k is a multiple of 5. At the same time, the results of the multiplications of any two remainders must be different, that is, for any distinct $a, b, c, \in 1 \dots 4$, if $ab = d$ and $ac = e$, then $d \neq e$. Otherwise, ab and ac would leave the same remainder with 5, which cannot be, since that would mean that there was a number k, such that $5k + ab = 5k + ac$, boiling down to $ab = ac$ and, by dividing a, $b = c$.

This is a significant result. It implies that, if we could devise an algorithm that finds the inverse of any $a \bmod n$ (for n being prime), we would not only have a division algorithm, but we would have defined a multiplication group over natural numbers – and this is where arithmetic modulo a prime is different from arithmetic modulo a composite.

We find such an algorithm if we go to the heart of the matter. It is related to a special property of gcd. We know that gcd proceeds by applying the modulo operation: $\gcd(a,b) = \gcd(b, a \bmod b)$. If c is the remainder, *i.e.* $c = a \bmod b$, then we have

$$c = a - qb, \tag{5.10}$$

where q is the quotient, *i.e.* the greatest number that multiplied with b is equal or less than a. This equation is equivalent to the following:

$$c = ka + lb, \tag{5.11}$$

where $k = 1$ and $l = -q$. We now prove by induction on gcd that, for any d such that $d = \gcd(a,b)$, there are two integers k and l, positive or negative, for which holds $d = ka + lb$. The base case is equation 5.11. We have to prove that, if 5.11 shows the n^{th} recursion step of gcd, then, in the $(n+1)^{th}$ rescursion step, it still holds for the remainder of the arguments in that iteration, d, that there are two integers m and n, such that $d = ma + nb$. Since the final result of gcd is the remainder of the previous recursion step, this proves that there are always two integers k and l such that $\gcd(a,b) = ka + la$.

If c in the equation $c = ka + lb$ represents the remainder in the n^{th} recursion step of *gcd*, then c is the second argument in the next recursion step and d in $d = b - qc$ represents the remainder in this step. We substitute the base case 5.11 for c:

$$d = b - qc = b - q(ka + lb) \tag{5.12}$$

Let us distinguish the quotient in equation 5.10 and the one in 5.12 by adding the subscripts: $c = a - q_1b$ and $d = b - q_2c$. Since $c = a - q_1b$, we can state that $d = b - q_2(a - q_1b)$ or, according to the base case 5.11:

$$d = b - q_2(ka + lb). \tag{5.13}$$

We multiply this out to get:

$$d = b - (q_2ka + q_2lb), \tag{5.14}$$

which is just

$$d = b - q_2ka - q_2lb. \tag{5.15}$$

By regrouping and adding $-q_2lb + b$, we obviously get

$$d = -q_2ka - (q_2l - 1)b. \tag{5.16}$$

We set $m = -q_2k$ and $n = -(q_2l - 1)$ and obtain the desired result:

$$d = ma + nb. \quad \Box \tag{5.17}$$

To illustrate this with an example, we claim that the remainder in the second iteration of $\gcd(21, 15)$ is $-21q_2k - 15(q_2l - 1)$, where k and l fulfil the equation $21k + 15l = (21 \bmod 15)$ and $l = -q_1$. the quotient of 21 and 15, which is 1. So we have $21k - 15 = 6$, which becomes true if $k = 1$.

In the next round, we have $\gcd(15, 6)$. The quotient of 15 and 6, q_2, is 2. We, hence, claim that $21 \times -2 \times 1 - 15 \times (2 \times -1 - 1) = 15 \bmod 6 = 3$. Let us see if this is true. We simplify to $-42 - 15(-2 - 1)$, which in its turn is $-42 - 15(-3)$ or $-42 + 45 = 3$, which is indeed the expected result.

Now we will look at the special case that the gcd of two numbers a and b is 1. There still must be two numbers k and l, such that

$$1 = ka + lb. \tag{5.18}$$

From this, we can prove as a corollary, that if a prime p divides ab, it must divide either a or b, a fact that we took for granted, when we proved the fundamental theorem of arithmetic. If p does not divide a, then we have $\gcd(a, p) = 1$ and, hence, $1 = ka + lp$. Multiplying by b, we get $b = b(ka + lp)$ or $b = kab + lbp$. The fact that p divides ab means that there is a number r, such that $ab = rp$. So, we can also say $b = krp + lbp$ or $b = p(kr + lb)$, where p clearly appears as a factor of b. $\Box$

Let us come back to our division problem. The relevant information is equation 5.18, *i.e.* that, if $\gcd(a, b) = 1$, then there are two integers k and l, such that $1 = ka + lb$. We want to get something for a that looks like $\frac{1}{a}$, since $a \times \frac{1}{a} = 1$. In other words: $\frac{1}{a}$ is the inverse of a. A function that would serve as such an inverse for a would be $f(x) = kx + lb$, for the two integers k and l.

If we compute $\gcd(a, n)$, where n is a prime, we know we get 1 back and we know there must be two integers k and l such that $1 = ka + ln$. We can transform this equation by subtracting ln to $ka = 1 - ln$. Since ln is a multiple of n, $1 - ln$, which is the same as $-ln + 1$, would leave the remainder 1 on division by n, *i.e.* $-ln + 1 = 1 \mod n$. In other words: $ka = 1 \mod n$. That is actually what we are looking for: a number that, multiplied by a, is 1. k, hence, is the wanted inverse of a modulo n. The question now is: how to get to k?

There is a well known algorithm that produces not only the greatest common divisor, but also k and l. This algorithm is called the *extended greatest common divisor* or xGCD:

```
xgcd :: Integer → Integer → (Integer, (Integer, Integer))
xgcd a b = go a b 1 0 0 1
  where go c 0 uc vc _ _   = (c, (uc, vc))
        go c d uc vc ud vd = let (q, r) = c `quotRem` d
                             in go d r ud vd (uc − q ∗ ud)
                                             (vc − q ∗ vd)
```

The listing, admittedly, looks somewhat confusing at the first sight. However, it bears the classic *gcd*. If you ignore the four additional parameters of *go*, you see that *go* calls *quotRem*, instead of just *rem* as *gcd* does, and it then recurses with *go d r*, d being initially b and r being the remainder – that is just *gcd*. But *go* additionally computes $uc - q \times ud$ and $vc - q \times vd$. We start with $uc = 1$, $vc = 0$, $ud = 0$ and $vd = 1$, hence: $1 - q \times 0 = 1$ and $0 - q \times 1 = -q$. These are just k and l after the first iteration. In the next iteration, we will have $uc = 0$, $vc = 1$, $ud = 1$ and $vd = -q_1$ and compute $0 - q_2 \times 1$ and $1 - q_1 \times -q_2$, which you will recognise as m and n from equation 5.16. The algorithm is just another formulation of the proof we have discussed above.

Let us look at *xgcd* with the example above, $a = 21$ and $b = 15$. We start to call *go* as *go* 21 15 1 0 0 1, which is

$(1, 6) = 21$ `quotRem` 15

in

go 15 6 0 1 $(1 - 1 * 0)$ $(0 - q * 1)$
go 15 6 0 1 1 (-1).

This leads to

$(2, 3) = 15$ `quotRem` 6

in

go 6 3 1 (-1) $(0 - 2 * 1)$ $(1 - 2 * (-1))$
go 6 3 1 (-1) (-2) 3.

In the next round we have

$(2, 0) = 6$ `quotRem` 3

and we now call, ignoring ud and vd:

go 3 0 (-2) 3 _ _

and, since $d = 0$, just yield $(3, (-2, 3))$, where the k we are looking for is -2, *i.e.* the first of the inner tuple.

Since 15 and 21 in the example above are not coprime the remainder is not 1 but 3 (since $\gcd(21, 15) = 3$), When we use the function with a prime number p and any number $a < p$, the remainder is 1. The resulting k is the inverse of a mod p and we can therefore use this k to implement division. But, actually, we are not in Kansas anymore: k may be negative. That is, even if we are still discussing problems of natural numbers, we have to refer to negative numbers and, thus, use a number type we have not yet implemented.

Technically, this is quite simple – it hurts of course that we have to cheat in this way. Anyway, here is a simple solution:

```
nxgcd :: Natural → Natural → (Natural, Natural)
nxgcd a n = let a'          = fromIntegral a
                n'          = fromIntegral n
                (r, (k, _)) = xgcd a' n'
            in if k < 0 then (fromIntegral r, fromIntegral (k + n'))
                        else (fromIntegral r, fromIntegral k)
```

This function is somewhat difficult to look through because of all the conversions. First we have to convert the natural numbers to integers using *fromIntegral*, then we have to convert the result back to a natural number, again, using *fromIntegral*. This works because both types, *Integer* and *Natural*, belong to class *Integral*.

After conversion, we apply *xgcd* on the integers ignoring l, just using the remainder and k. (The reason that we do not throw away the remainder as well is that we will need the remainder sometimes to check that $xgcd\ a\ b \equiv 1$.) Now, if k is a negative number, we add the modulus n to it, as we have learnt, when we studied subtraction.

Let us specialise *nxgcd* for the case that we only want to have the inverse:

```
inverse :: Natural → Natural → Natural
inverse a = snd ∘ nxgcd a
```

The inverses of the numbers modulo 5 are for instance:

$1 : inverse\ 1\ 5 = 1$
$2 : inverse\ 2\ 5 = 3$
$3 : inverse\ 3\ 5 = 2$
$4 : inverse\ 4\ 5 = 4.$

Note that there is no inverse for 0, since any number multiplied by 0 is just 0. Another way to state this is that 1/0 is undefined.

Any other number a and its inverse a' behave as follows: $a \times a' = 1$ and, of course, $a \times b \times a' = b$. For instance:

$1 \times 1 = 1 \mod 5$
$2 \times 3 = 1 \mod 5$
$3 \times 2 = 1 \mod 5$
$4 \times 4 = 1 \mod 5.$

We can also play around like:

$2 \times 4 \times 3 = 4 \mod 5$
$3 \times 4 \times 4 = 3 \mod 5$
$3 \times 1001 \times 2 = 1001 = 1 \mod 5.$

Division, the inverse operation to multiplication, is now easily implemented as:

```
mDiv :: Module → Module → Module
mDiv (Module n a1) (Module _ a2) = Module n (((inverse a2 n) ∗ a1) ‘rem‘ n)
```

With this function, we have a way to make *Module* member of the *Integral* class, but, before we can do that, we have to make it instance of the *Enum* and the *Real* classes, which is straight forward:

```
instance Enum Module where
  fromEnum (Module _ a) = fromIntegral a
  toEnum i              = tomod (fromIntegral (i + 1)) (fromIntegral i)
instance Real Module where
  toRational (Module _ a) = fromIntegral a
```

The *Integral* instance is now simply defined as:

```
instance Integral Module where
  quotRem x@(Module n _) y = (withGuard mDiv x y, Module n 0)
  toInteger (Module _ a)   = fromIntegral a
```

For *quotRem*, *mDiv* is used to compute the quotient and, since we have defined division in terms of multiplication, we know that there is never to be a remainder different from 0. We, hence, just return a *Module* with the value 0 as remainder of *quotRem*.

We could now go even further and define an instance for *Fractional*:

```
instance Fractional Module where
  (/) = mDiv
  fromRational = ⊥
```

It is nice to have the division operator available for modules, so we can do things like *a* / *b*, where *a* and *b* are of type *Module*. For *fromRational*, however, which is mandatory for defining the *Fractional* class, we have, for the time being, no good implementation.

There is an important corollary that follows from the invertibility of numbers modulo a prime, namely that any number in the range $1 \dots p-1$ can be created by multiplication of other numbers in this range and for any two number a and n, there is unique number b that fulfils the equation

$$ax = n. \tag{5.19}$$

In other words: There are no primes modular a prime. For natural numbers with ordinary arithmetic, this is clearly not true. There is for instance no solution for equations like $3x = 2$ or $3x = 5$. In arithmetic modulo a prime, however, you always find a solution, for instance: $3x = 2 \mod 5$ has the solution 4, since $3 \times 4 = 12 = 2 \bmod 5$.

This follows immediately from invertibility, since we only have to multiply n to the inverse of a to find x. If we have the inverse a' of a, such that

$$aa' = 1, \tag{5.20}$$

we just multiply n on both sides and get:

$$naa' = 1n. \tag{5.21}$$

For the example above, $3x = 2 \mod 5$, we can infer x from

$$3 \times 2 = 1 \mod 5 \tag{5.22}$$

by multiplying 2 on both sides:

$$2 \times 3 \times 2 = 2 \mod 5. \tag{5.23}$$

$ax = 2 \mod 5$, hence, has the solution $x = 4$. Here is a kind of magic square for numbers modulo 5:

	1	2	3	4
1	1	3	2	4
2	2	1	4	3
3	3	4	1	2
4	4	2	3	1

The leftmost column shows a multiplication result. The multiplication is defined as: $row_1 \times row_n$. The second row, with 1 in the first column, shows the inverse for each number: The inverse of 1 is 1; the inverse of 2 is 3; the inverse of 3 is 2 and the inverse of 4 is 4. The next row shows the multiplications resulting in 2: 1×2, 2×1, 3×4 and 4×3.

Let us summarise what we have learnt. Arithmetic modulo a prime p constitutes a finite field of the numbers $0 \dots p-1$. The arithmetic operations on numbers modulo p always yield a number in that range, *i.e.* the operations are closed modulo p. Additionally to the properties we had already seen for natural numbers, associativity, identity and commutativity, we saw that operations modulo p are invertible for both addition and multiplication. In spite of the observation that all numbers we are dealing with are positive integers, *i.e.* natural numbers, subtraction and division are closed and every number modulo a prime has an inverse number for addition and multiplication. For the multiplicative group of the field, 0 must be excluded, since there is no number k such that $0 \times k = 1$ or, stated differently, $1/0$ is undefined. This, however, is true for all multiplicative groups.

5.7. Congruence

There is an important fact that, in the light of modular arithmetic, appears to be completely trivial, namely that all numbers $0 \dots n-1$ leave the same remainder divided by n as infinitely many other numbers $\geq n$. For instance, 0 leaves the same remainder as n divided by n; 1 leaves the same remainder as $n+1$; 2 leaves the same remainder as $n+2$ and so on. Furthermore, 1 leaves the same remainder as $2n+1$, $3n+1$, $4n+1$, ... This relation, that two numbers leave the same remainder divided by another number n, is called congruence and is written:

$$a \equiv b \pmod{n}.$$

We have for example:

$$1 \equiv mn + 1 \pmod{n} \tag{5.24}$$

$$2 \equiv mn + 2 \pmod{n} \tag{5.25}$$

$$k \equiv mn + k \pmod{n} \tag{5.26}$$

An important congruence system is Fermat's *Little Theorem*, which is called like this to distinguish it from the other famous theorem by Pierre Fermat, his *Last Theorem*, which, in its turn, is named this way, because, for many centuries, it was the last of Fermat's propositions that was not yet proven.

Fermat's little theorem states that, for any integer a and any prime number p:

$$a^p \equiv a \pmod{p}, \tag{5.27}$$

which is the same as:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{5.28}$$

That the two equations are equivalent is seen immediately, when we multiply both sides of the second equation with a: $a \times a^{p-1} = a^p \equiv a \times 1 = a \pmod{p}$.

One of the proofs of the little theorem brings two major themes together that we have already discussed, namely binomial coefficients and modular arithmetic. You may have observed already that all binomial coefficients $\binom{p}{k}$, where p is prime and $0 < k < p$, are multiples of p. For instance:

$\binom{3}{2} = 3$
$\binom{5}{2} = 10$
$\binom{5}{3} = 10$
$\binom{7}{2} = 21$
$\binom{7}{3} = 35$
$\binom{7}{4} = 35$
$\binom{7}{5} = 21.$

This is not true for coefficients where p is not prime. For instance:

$\binom{4}{2} = 6$
$\binom{6}{2} = 15$
$\binom{8}{2} = 28$
$\binom{8}{4} = 70.$

To prove this, we first observe that all binomial coefficients are integers. Since we hardly know anything but natural numbers, we will not prove this fact here, but postpone the discussion to the next chapter. One way to define binomial coefficients is by means of factorials:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{5.29}$$

We transform this equation multiplying $k!(n-k)!$ on both sides:

$$n! = \binom{n}{k} k!(n-k)!, \tag{5.30}$$

which shows that n must divide either $\binom{n}{k}$ or $k!(n-k)!$, because the product of these two factors equals $n!$, which is a multiple of n by definition. Note that this is just the application of Euclid's lemma to a prime number. Since a prime has no factors to share with other numbers but itself, it must divide at least one of the factors of a product that it divides.

Let us check if n divides $k!(n-k)!$ Again, to divide the whole, n must divide one of the factors, either $k!$ or $(n-k)!$ But, if n is prime and $0 < k < n$ and $0 < n-k < n$, it cannot divide either of them, since none of the factors of $k!$ $(1 \times 2 \times \cdots \times k)$ and none of the factors of $(n-k)!$ $(1 \times 2 \times \cdots \times (n-k))$ is divided by n or divides n. One cannot compose a number that is divisible by a prime by multiplying only numbers that are smaller than that prime. We could do so easily for composites. $4! = 24$, for instance, is divisible by 8. But no number smaller than a given prime multiplied by another number smaller than that prime, will ever be divided by that prime. So, obviously, n must divide $\binom{n}{k}$ or, in other words, $\binom{n}{k}$ is a multiple of n. □

To the delight of every newcomer, it follows immediately from this fact that, if p is prime:

$$(a+b)^p \equiv a^p + b^p \pmod{p}. \tag{5.31}$$

This identity, for understandable reasons, is sometimes called *Freshman's Dream.* The binomial theorem, which we have already proven, states:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}. \tag{5.32}$$

Since all $\binom{n}{k}$ for $0 < k < n$ are multiples of n if n is prime, they are all $0 \bmod n$. (Remember that, in modular arithmetic, we can take the modulo at any point when calculating a complex formula!) In the summation, hence, all terms for the steps $0 < k < n$ are 0 and only the first case, $k = 0$, and the last case, $k = n$, remain, whose coefficients are $\binom{n}{0} = 1$ and $\binom{n}{n} = 1$. The resulting formula, hence, is

$$(a+b)^n = \binom{n}{n} a^n + \binom{n}{0} b^{n-0} = a^n + b^n \quad \square$$

We will now prove Fermat's little theorem by induction. We choose the base case $a = 1$. Since $1^p = 1$, it trivially holds that $1^p \equiv 1 \pmod{p}$. We now have to prove that, if $a^p \equiv a \pmod{p}$ holds, it also holds that

$$(a+1)^p \equiv a+1 \pmod{p}. \tag{5.33}$$

$(a+1)^p$ is a binomial formula with a prime exponent. We have already shown that $(a+b)^p = a^p + b^p \mod p$ and we can therefore conlcude $(a+1)^p = a^p + 1^p \mod p$, which, of course, is just $a^p + 1$. From the base case we know that $a^p \equiv a \pmod{p}$ and can therefore further conclude that $a^p + 1 \equiv a + 1 \pmod{p}$. □

Another interesting congruence system with tremendous importance in cryptography is the *Chinese Remainder Theorem.* The funny name results from the fact that systems related to this theorem were first investigated by Chinese mathematicians, namely Sun Tzu, who lived between the 3rd and the 5th century, and Qin Jiushao, who provided a complete solution in his “Mathematical Treatise in Nine Sections” published in the mid-13th century.

The theorem deals with problems of congruence systems where the task is to find a number x that leaves given remainders with given numbers. We can state such systems in general as follows:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\dots \\ x &\equiv a_r \pmod{n_r} \end{aligned}$$

The theorem now states for x, $a_1 \dots a_r$ and $n_1 \dots n_r \in \mathbb{N}$ that, if the numbers $n_1 \dots n_r$ are coprime, then there is always a solution for x, which even further is unique modulo $\prod n_i$, the product of all the ns and, since the ns are coprime to each other, their least common multiplier.

Before we prove this theorem, let us look at potential algorithms to solve such systems. The first account would be "common sense": we would just search "brute-force" for the proper solutions among candidates. Candidates are all numbers congruent to $a_1 \dots a_r$ modulo $n_1 \dots n_r$. For each pair of (a_i, n_i), we would create a list of congruences. Solutions would be the numbers that are in all such lists, *i.e.* the intersection of those lists. We would start by creating lists of congruences. The first element in the list of congruences for a pair (a_i, n_i) would be a_i, the next would be $a_i + n_i$ (since that number leaves the same remainder as a_i divided by n_i):

```
congruences :: Natural → Natural → [Natural]
congruences a n = a : congruences (a + n) n
```

This function will create an infinite list of all numbers leaving the same remainder with n (which is n_i) as a (which is a_i). For $a = 2$ and $n = 3$, *i.e.* the congruence $x \equiv 2 \pmod 3$, we would generate the list:

$[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, ..]$

We would then devise a function to apply this generator to all pairs of (a_i, n_i) in the congruence system at hand:

```
mapCongruences :: Int → [Natural] → [Natural] → [[Natural]]
mapCongruences l as ns = [take l (congruences a n) | (a, n) ← zip as ns]
```

This function receives three arguments: l of type *Int* (which should rather be *Natural*, but *Int* is chosen for convenience, since it is used with *take*) and *as* and *ns* both of type [*Natural*]. The function simply applies all pairs of (a, n) to the *congruences* generator. It limits the length of resulting lists from the generator to l. Otherwise, the list comprehension would never come to a result that we could then use to find the solution. Calling *mapCongruences* for the simple congruence system

$$\begin{aligned} x &\equiv 2 \pmod{3} \\ x &\equiv 3 \pmod{4} \\ x &\equiv 1 \pmod{5} \end{aligned}$$

with $l = 10$ yields three lists:

$[2, 5, 8, 11, 14, 17, 20, 23, 26, 29]$
$[3, 7, 11, 15, 19, 23, 27, 31, 35, 39]$
$[1, 6, 11, 16, 21, 26, 31, 36, 41, 46]$.

Now we just have to intersect these lists:

```
chinese1 :: [Natural] → [Natural] → [Natural]
chinese1 as ns = case mapCongruences (fromIntegral $ product ns) as ns of
                   [] → []
                   cs → foldr intersect (head cs) (tail cs)
```

This function receives two arguments, the list of $a_1 \dots a_r$ and the list of $n_1 \dots n_r$. On these lists, it calls *mapCongruences* with $l = \prod n_i$. The idea behind this choice will become clear in a minute. If the result of this application is an empty list, we return an empty list. This is just a trick to avoid an exception in the case where the input consists of empty lists. Otherwise, we fold the result list with *intersect* and *head cs* as the base case for *foldr*. The result for *chinese1* $[2, 3, 1]$ $[3, 4, 5]$ is

$[11, 71, 131]$.

If the theorem is correct, the three numbers in the resulting list should be congruent to each other modulo $3 \times 4 \times 5 = 60$. The call *map* (‘*rem*‘60) $[11, 71, 131]$, indeed, yields $[11, 11, 11]$.

The fact that the solution is unique modulo the product of all ns implies that there must be at least one solution in the range $a_s \cdots \prod n_i$, where a_s is the smallest of the as in the system. There is not necessarily a solution in the range of any particular n. But if there was no solution less than the product $\prod n_i$, there trivially would be no solution at all.

To guarantee that we look at all candidates up to $\prod n_i$, we take lists of the length $\prod n_i$. This is certainly exaggerated, since in lists with that number of elements, there are already much greater numbers. But it guarantees that we will find a solution.

This brute-force algorithm is quite instructive, since it shows the structure of the problem quite well. On the other hand, it is inefficient in terms of computational complexity. For big systems and, in particular, for large ns the congruence lists become unmanageably large. We should look for an algorithm that exploits our knowledge on modular arithmetic.

To start with, we observe that, obviously, all ns divide $\prod n_i$ (which we will call pN in the remainder of this section). But, since the ns are coprime to each other, n_i would not divide the product of all numbers but itself, that is $\frac{pN}{n_i}$. In other words $gcd(\frac{pN}{n_i}, n_i) = 1$. There, hence, exist two integers, k and l, such that $k \times \frac{pN}{n_i} + ln_i = 1$. Since ln_i is a multiplie of n_i, this means that $k \times \frac{pN}{n_i} = 1 \bmod n_i$. The number k is thus the inverse of $\frac{pN}{n_i} \bmod n_i$. Let us call the product of k and $\frac{pN}{n_i}$ (of which we know that it is 1 mod n_i) e_i: $e_i = k \times \frac{pN}{n_i}$.

We could now write the ridiculous formula $x \equiv e_i \times a_i \pmod{n_i}$ where we multiply e_i with a_i in the corresponding line of the congruence system. The formula is ridiculous, because we already know that $e_i \bmod n_i = 1$, the formula, hence, says $x \equiv 1 \times a_i \pmod{n_i}$, which adds very little to the original formulation $x \equiv a_i \pmod{n_i}$. But, actually, this stupid formula leads directly to the solution.

Note that for any $n_j, j \neq i$, n_j divides $\frac{pN}{n_i}$ and all its multiples including $e_i = k \times \frac{pN}{n_i}$. That is, for any e_i: $e_i \equiv 0 \pmod{n_j}, j \neq i$. So we could create the following, equally ridiculous equation:

$$x \equiv \sum_{j=1}^{r} e_j \times a_i \pmod{n_i}. \tag{5.34}$$

Since, for any specific i, all $e_j, j \neq i$, are actually 0 and for the one case, where $j = i$, e_i is 1, this equation is trivially true for any line in the system. It just states $x \equiv a_i \pmod{n_i}$. In other words: this sum fits all the single lines of the congruence system.

This trivially magic sum not taken modulo to any of the individual ns is of course a number that is much larger – or, much smaller, *i.e.* a large negative number – than the smallest number that would fulfil all congruences in the system. The unique solution, however, is this number taken modulo pN. Since pN is just a multiple of any of the n_i in the system, all numbers leaving the same remainder modulo pN will leave the same remainder modulo a specific n_i.

This approach to Chinese remainder systems is much more efficient than the brute-force logic we implemented before. To implement it in Haskell, we first implement the function that finds e_i, using the extended gcd:

```
inv :: Integer -> Integer -> Integer
inv n pN = let b             = pN `div` n
               (_, (k, _)) = xgcd b n
           in  k * b
```

Then we call this function for each pair (a_i, n_i) in the system, sum the products $a_i \times e_i$ and yield the result modulo pN:

```
chinese :: [Integer] → [Integer] → Natural
chinese as ns = let pN = product ns
                    es  =         [inv n pN | n      ← ns]
                    e   = sum [a ∗ e      | (a, e) ← zip as es]
                in  fromIntegral (e ‘nmod‘ pN)
```

Since we are working with integers here instead of natural numbers – giving up to pretend that we can solve all problems related to natural numbers with natural numbers alone – we use a *mod* operator that is modelled on the *Module* data type defined in the previous section:

```
nmod :: Integer → Integer → Integer
nmod x n | x < 0     = n − ((−x) ‘rem‘ n)
         | otherwise = x ‘rem‘ n
```

Consider the example we already used above:

$$\begin{aligned} x &\equiv 2 \pmod{3} \\ x &\equiv 3 \pmod{4} \\ x &\equiv 1 \pmod{5}. \end{aligned}$$

We would solve this system by calling *chinese* $[2,3,1]$ $[3,4,5]$. The results for *inv* are:

$\mathit{inv}\ 3\ 60 = -20$
$\mathit{inv}\ 4\ 60 = -15$
$\mathit{inv}\ 5\ 60 = -24.$

We would now call:

$\mathit{sum}\ [2 * (-20), 3 * (-15), 1 * (-24)] = \mathit{sum}\ [-40, -45, -24] = -109$

and take the result modulo 60: (-109) ‘*nmod*‘ $60 = 11$. Confirm that this result fulfils the system:

$$\begin{aligned} 11 &\equiv 2 \pmod{3} \\ 11 &\equiv 3 \pmod{4} \\ 11 &\equiv 1 \pmod{5}. \end{aligned}$$

5.8. Quadratic Residues

Quadratic residues of a number n are natural numbers congruent to a perfect square modulo n. In general, q is a quadratic residue modulo n if:

$$x^2 \equiv q \pmod{n}. \tag{5.35}$$

A simple function to test whether a number q is indeed a quadratic residue with respect to another number x could look like this:

```
isResidue :: Natural → Natural → Natural → Bool
isResidue q n x = (x ↑ 2) ‘rem‘ n ≡ q
```

This function is a nice test, but it does not help us to find residues. The following function does that:

```
residues :: Natural → [Natural]
residues n = sort (nub [(x ↑ 2) ‘rem‘ n | x ← [0 . . n − 1]])
```

residues finds all residues modulo n by simply taking the remainder of the squares of all numbers $0 \dots n-1$. Note that these are all remainders of squares modulo this number. Any other square, for instance the square $(n+1)^2$, will reduce to one of the remainders in the range $0 \dots n-1$. $(n+1)^2$ would just reduce to the remainder 1; $(n+2)^2$ would reduce to the remainder 2; likewise $(mn+1)^2$ would reduce to 1 or, in general, any number of the form $(mn+r)^2$, where r is a number from the range $0 \dots n-1$, will always reduce to r.

Since *residues* tests all numbers in the range $0 \dots n-1$, some numbers may appear more than once. The residues of 6, for instance, are: $0, 1, 4, 3, 4, 1$, since

$$\begin{aligned}
0^2 &= 0 \equiv 0 \pmod{6}\\
1^2 &= 1 \equiv 1 \pmod{6}\\
2^2 &= 4 \equiv 4 \pmod{6}\\
3^2 &= 9 \equiv 3 \pmod{6}\\
4^2 &= 16 \equiv 4 \pmod{6}\\
5^2 &= 25 \equiv 1 \pmod{6}
\end{aligned}$$

The function, therefore, *nub*s the result and sorts it for convenience.

Let us look at the residues of some small numbers:

residues $9 = [0, 1, 4, 7]$
residues $15 = [0, 1, 4, 6, 9, 10]$
residues $21 = [0, 1, 4, 7, 9, 15, 16, 18]$

What happens, when the modulus is prime? Some examples:

residues $3 = [0, 1]$
residues $5 = [0, 1, 4]$

residues $7 = [0, 1, 2, 4]$
residues $11 = [0, 1, 3, 4, 5, 9]$
residues $13 = [0, 1, 3, 4, 9, 10, 12]$

Apparently, the number of residues per modulus is constantly growing. In the case of prime moduli, however, there appears to be a strict relation between the modulus and the number of residues. There seem to be roughly $p/2$ residues for a prime modulus p or, more precisely, there are $(p + 1)/2$ residues (if we include 0). This is not the fact with composite moduli. Let us devise a function that may help us to further investigate this fact:

```
countResidues :: Natural → Int
countResidues = length ∘ residues
```

Applied to a random sequence of composite numbers the result appears to be random too (besides the fact that the number of residues is slowly growing together with the moduli):

9	15	21	25	26	30	32	35	90	100	150	500
4	6	8	11	14	12	7	12	24	22	44	106

Applied on prime numbers the result is always $(p + 1)/2$:

3	5	7	11	13	17	19	23	29	31	37	41
2	3	4	6	7	9	10	12	15	16	19	21

and so on. There is, however, one remarkable exception, namely 2: *residues* $2 = [0, 1]$ and, hence, *countResidues* $2 = 2$. The general rule is therefore that, for an **odd** prime p, there are $(p + 1)/2$ residues and $(p - 1)/2$ nonresidues.

When we look at the residues of primes above, we see that some numbers appear more than once. For instance, 4 is residue of 5, 7, 11 and 13; 2, by contrast, appears only once; 3 appears twice. An interesting line of investigation could be, which prime moduli have a certain residue and which have not. The following function is a nice tool for this investigation:

```
hasResidue :: Integer → Integer → Bool
hasResidue n q    | q < 0 ∧ abs q > n  = hasResidue n (q ‘rem‘ n)
                  | q < 0              = hasResidue n (n + q)
                  | q ≡ 0              = True
                  | otherwise          = check 0
   where check x  | x ≡ n              = False
                  | (x ↑ 2) ‘rem‘ n ≡ q = True
                  | otherwise          = check (x + 1)
```

There is something special about this function that should be explained. First thing to notice is that it does not operate on *Natural*, but on *Integer* and, indeed, the first two guards immediately take care of negative residues (q). In the first line, a negative q with an absolute value greater than n is reduced to the negative remainder; for a negative remainder already reduced to a remainder modulo n, the function is simply called again on $n + q$, that is n minus the absolute value of q. This is *negative congruence* that we already encountered, when we started to discuss arithmetic modulo a prime. It is a way to generalise the case of $n - a$, where we are not looking for a fixed number, but for a residue defined relative to n, *e.g.* $n - 1$, which would just be -1.

For $q = 0$, the function simply yields *True*, since 0 is residue of any number. For positive integers, *hasResidue* calls *check* 0. *check*, as can be seen in the third line, counts the $x$es up to n. When it reaches n, it yields *False* (first line). Should it encounter a case where x^2 equals q mod n, it terminates yielding *True*.

We can test the function asking for 4 in 9, 15 and 21 and will see that in all three cases, the result is *True*. Now we would like to extend this to learn if all numbers starting from 5 (where it appears for the first time) have the residue 4:

```
haveResidue :: Integer → [Integer] → [Integer]
haveResidue _ [ ] = [ ]
haveResidue q (n : ns) | n ‘hasResidue‘ q = n : haveResidue q ns
                       | otherwise        =     haveResidue q ns
```

This function searches for numbers in a given list (second argument) that have q as a residue. We could, for instance, call this function on all numbers from 5 onwards (restricting the result to 10): *take* 10 (*haveResidue* 4 [5 ..]). The result is indeed $[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$.

Do not let yourself be confused by the fact that 4 is a quadratic residue of all numbers greater than 4. It is just trivial; in fact, all perfect squares are residues of numbers greater than these squares. The same is true for 9, which is residue of $10, 11, 12, \dots$; 16 is residue of $17, 18, 19, \dots$ and in general any number of the form x^2 is residue of any number $n > x^2$. This is just the definition of quadratic residue: $x^2 \equiv q \pmod{n}$, for the special case where $q = x^2$, which is trivially true, whenever $n > x^2$.

To continue the investigation into residues of primes, we specialise *haveResidue* to odd primes:

```
primesWithResidue :: Integer → [Integer]
primesWithResidue = (‘haveResidue‘(drop 1 intAllprimes))
  where intAllprimes = map fromIntegral allprimes
```

Since we have already seen that all numbers from 5 on have 4 as a residue, *primesWithResidue* 4, will just give us the primes starting from 5. A more interesting investigation is in fact -1, *i.e.* all primes p that have $p - 1$ as a residue:

primesWithResidue $(-1) = [5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97 ..]$.

Is there something special about this list of primes? Not, perhaps, on the first sight. However, try this: *map* (‘*rem*‘4) (*primesWithResidue* (-1)) and you will see an endless list: $1, 1, 1, 1, 1, 1, \ldots$. In other words, all these primes are $\equiv 1 \pmod 4$. If this is true, the following function should create exactly the same list of primes:

```
minus1Residues :: [Integer]
minus1Residues = go (drop 1 intAllprimes)
  where go []  = []
        go (p : ps) | p `rem` 4 ≡ 1 = p : go ps
                    | otherwise     =     go ps
        intAllprimes = map fromIntegral allprimes
```

And, indeed, it does:

minus1Residues $= [5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97\,..]$.

This fact is quite important. It is known as the *first supplement* to the *law of quadratic reciprocity.* We will start the proof of the first supplement with an apparently unrelated theorem, namely *Wilson's Theorem*, which states that if and only if n is prime, then it holds that

$$(n-1)! \equiv -1 \pmod n. \tag{5.36}$$

To check this quickly with some small primes:

$$\begin{aligned} 2! = 2 &\equiv -1 \pmod 3 \\ 4! = 24 &\equiv -1 \pmod 5 \\ 6! = 720 &\equiv -1 \pmod 7 \\ 10! = 3628800 &\equiv -1 \pmod{11} \end{aligned}$$

and some small composites:

$$\begin{aligned} 3! = 6 &\equiv 2 \pmod 4 \\ 5! = 120 &\equiv 0 \pmod 6 \\ 7! = 5040 &\equiv 0 \pmod 8 \end{aligned}$$

The proof is rather simple. We first prove that Wilson's theorem holds for all primes, then we prove that it does not hold for composites.

To prove that it holds for primes, remember from modular arithmetic that, with a prime modulus n, every number $a \in 1 \ldots n-1$ has an inverse $a' \in 1 \ldots n-1$, such that $a \times a' = 1$ mod n. For instance, the numbers $1 \ldots 6$ and their inverses modulo 7 are:

$$(1,1),(2,4),(3,5),(6,6)$$

For all pairs of numbers (a,a') where $a \neq a'$, the product of the pair is just 1. The example above seems to suggest that for all numbers, but 1 and $n-1$, $a \neq a'$. If this is true, then the factorial of $n-1$ modulo n, where n is prime, would translate into $1 \times 1 \times \dots \times n-1$, which obviously is $\equiv (n-1) \pmod{n}$. But does it actually hold for all primes? Let us look: if a is its own inverse, *i.e.* $a = a'$, we would have $a^2 \equiv 1 \pmod{n}$. In other words: a^2 would leave a remainder of 1 divided by n. a^2-1, hence, should leave no remainder, thus: $a^2 - 1 \equiv 0 \pmod{n}$. We can factor a^2-1 into $(a+1)(a-1)$. The factors help us to find numbers that substituted for a would make the whole expression 0. One possibility is obviously 1: $(1+1) \times (1-1) = 2 \times 0 = 0$. Another possibility, however, is $n-1$:

$$(n-1+1)(n-1-1) =$$
$$n(n-2) = n^2 - 2n$$

and, since $n^2 - 2n$ contains only multiples of n:

$$n^2 - 2n \equiv 0 \pmod{n}.$$

This does not hold for any other number from the range $2 \dots n-1$, since there will be always a remainder that does not reduce to a multiple of n, for instance: $(n-2+1)(n-2-1) = (n-1)(n-3) = n^2 - 4n + 3$, which is congruent to $3 \pmod{n}$; $(2+1)(2-1) = 3 \times 1 = 3$, which, again, is congruent to $3 \pmod{n}$ and in general $(n-a+1)(n-a-1) = n^2 - 3an + a^2 - 1$, which, modulo n, is $a^2 - 1$. If $a \neq 1$, this is not congruent $0 \pmod{n}$. It therefore holds for all primes n that $(n-1)! \equiv -1 \pmod{n}$. $\square$

Now the second part of the proof: That Wilson's theorem is never true when n is composite. We prove by contradiction and assume that there is a composite n, such that $(n-1)! \equiv -1 \pmod{n}$. That n is composite means that there is a prime number p that divides n. This p is one of the prime factors of n, *i.e.*: $n = mp$, for some integer m. This also means that p is smaller than n and in the range $2 \dots n-1$. Therefore, p must also divide $(n-1)!$, because it appears as one of the factors in $1 \times 2 \times \dots \times n-1$. In other words: $(n-1)! \equiv 0 \pmod{p}$.

But we also have $n = mp$. From modular arithmetic we know that if $a \equiv b \pmod{n}$, then also: $a \equiv b \pmod{mn}$. So, from $(n-1)! \equiv 0 \pmod{p}$, it follows that also $(n-1)! \equiv 0 \pmod{mp}$ and, since $n = mp$, $(n-1)! \equiv 0 \pmod{n}$. This contradicts our assumption that there is a composite n, such that $(n-1) \equiv -1 \pmod{n}$. $\square$

Let us come back to the first supplement, which claims that -1 is a residue of an odd prime if and only if that prime is congruent 1 modulo 4. Any number not divided by 4 is either 1, 2 or 3 modulo 4. Since we are dealing only with odd primes, we can ignore the case 2, because no odd number will ever leave the remainder 2 divided by 4. That does only happen with even numbers not divided by 4, like 6, 10, 14, *etc.* We, hence, distinguish two cases: $p \equiv 1$ and $p \equiv 3$ both modulo 4. We first prove that an odd prime $p \equiv 1 \pmod 4$ has residue -1 and then that an odd prime $p \equiv 3 \pmod 4$ does not have residue -1.

We start with the observation that $p \equiv 1 \pmod 4$ implies $(p-1) \equiv 0 \pmod 4$, *i.e.* that $p-1$ is divisible by 4. This, in its turn, implies that we can group the remainders $1 \dots p-1$ into two sets with the same even number of elements. The remainders of the prime 5, for example, are $1, 2, 3, 4$ and we can group them into $\{\{1, 2\}, \{3, 4\}\}$. For 13, these groups would be $\{\{1, 2, 3, 4, 5, 6\}, \{7, 8, 9, 10, 11, 12\}\}$.

Now we rewrite the second group in terms of negative congruences:

$$\{\{1, 2, 3, 4, 5, 6\}, \{-6, -5, -4, -3, -2, -1\}\}$$

and then organise the groups as pairs of equal absolute values:

$$\{(1, -1), (2, -2), (3, -3), (4, -4), (5, -5), (6, -6)\}.$$

To compute the factorial of $p-1$, we could first multiply the members of each pair: $\{-1, -4, -9, -16, -25, -36\}$. It is essential to realise that the number of negative signs is even because the number of elements of each group is even. They, hence, cancel out on multiplication. This would be the same as squaring the members of the first group (1 to 6) before multiplying them: $\{1, 4, 9, 16, 25, 36\}$. We can do this the other way round as well: first, we multiply the two halfs out, creating the factorial for each group, and then multiply the two equal results, which is the same as squaring one of the results. In other words, if $p \equiv 1 \pmod 4$, then

$$(p-1)! = \left(\frac{p-1}{2}\right)!^2. \tag{5.37}$$

The right-hand side of this equation is a formal description of what we did above. We split the numbers $1 \dots p-1$ into halves: $1 \dots \frac{p-1}{2}$ and $-\frac{p-1}{2} \dots -1$, computed the factorial of each half and then multiplied the results, which of course are equal and multiplying them is thus equivalent to squaring.

For a prime p with the residue -1, there must be one number a, such that $a^2 \equiv -1 \pmod p$. The equation above shows that $(\frac{p-1}{2})!^2$ is actually $(p-1)!$, which, according to Wilson's theorem, is $-1 \pmod p$. $\frac{p-1}{2}!$, which squared is $(p-1)!$ and, according to Wilsons' theorem, -1. $\frac{p-1}{2}!$ is therefore such a number a. This, as shown above, is the case, if we can split the sequence of numbers $1 \dots p-1$ into two halves with an even number of members each. This, however, is only possible for an odd prime p, if $p-1 \equiv 0 \pmod 4$, which implies that $p \equiv 1 \pmod 4$. □

We will now show that primes of the form $p \equiv 3 \pmod 4$ do not have the residue -1 to complete the proof. Let us assume there is a number a, such that $a^2 \equiv -1 \pmod p$ and $p \equiv 3 \pmod 4$.

We start with the equation

$$a^2 \equiv -1 \pmod p \tag{5.38}$$

and raise both sides to the power of $\frac{p-1}{2}$:

$$a^{2\frac{p-1}{2}} \equiv -1^{\frac{p-1}{2}} \pmod p. \tag{5.39}$$

This can be simplified to

$$a^{p-1} \equiv -1^{\frac{p-1}{2}} \pmod p. \tag{5.40}$$

Note that $p-1$ is even (since p is an odd prime). But, since it is not divisisble by 4, since otherwise $p \equiv 1 \pmod 4$, $(p-1)/2$ must be odd. An example is $p = 7$, for which $(p-1)/2 = 3$. -1 raised to an odd power, however, is -1 and therefore we have:

$$a^{p-1} \equiv -1 \pmod p. \tag{5.41}$$

But that cannot be true, because Fermat's little theorem states that

$$a^{p-1} \equiv 1 \pmod p \tag{5.42}$$

There is only one p for which both equations are true at the same time, namely 2: $a^1 \equiv 1 \pmod 2$. This is actually true for any odd a.

But we are looking at odd primes and 2 is not an odd prime. Therefore, one of the equations must be wrong. Since we know for sure that Fermat's theorem is true, the wrong one must be 5.41. Therefore, -1 cannot be a residue of primes of the form $p \equiv 3 \pmod 4$ □.

There is also a *second supplement* to the law of reciprocity, which happens to deal with 2 as residue. When we look at these numbers, using *primesWithResidue* 2, we get: 7, 17, 23, 31, 41, 47, 71, 73, 79, 89, 97, 103,... What do these primes have in common? They are all one off numbers divisible by 8:

$$\begin{aligned}7 &\equiv -1 \pmod 8\\ 17 &\equiv 1 \pmod 8\\ 23 &\equiv -1 \pmod 8\\ 31 &\equiv -1 \pmod 8\\ 41 &\equiv 1 \pmod 8\\ 47 &\equiv -1 \pmod 8\\ &\dots\end{aligned}$$

The second supplement indeed states that ±2 is residue of an odd prime p if and only if $p \equiv \pm 1 \pmod 8$.

We will not prove this theorem here. Instead, we will look at a general criterion to decide quickly whether a number is residue of an odd prime, namely *Euler's Criterion*, which states that, if p is an odd prime, then:

$$a^{\frac{p-1}{2}} \equiv 1 \pmod p \text{ iff } a \text{ is a residue of } p \tag{5.43}$$

$$a^{\frac{p-1}{2}} \equiv -1 \pmod p \text{ iff } a \text{ is a nonresidue of } p \tag{5.44}$$

We can translate this criterion into Haskell as:

```
euCriterion :: Integer → Integer → Bool
euCriterion a p | a < 0 ∧ abs a > p = euCriterion (a `rem` p) p
                | a < 0             = euCriterion (p + a) p
                | otherwise         = let n = (p − 1) `div` 2
                                      in (a ↑ n) `rem` p ≡ 1
```

As we did before, we first handle the cases of negative congruence: a negative number is reduced to a negative number with an absolute value in the range of $1 \dots p-1$ and then p is added. A positive number is just raised to the power of $(p-1)/2$. If the remainder of this number is 1, this number is indeed a residue of p.

You see that in Euler's Criterion there appears a formula that we already know from the first supplement and, indeed, the proof of the Criterion with the background of the first supplement is quite simple.

We start by considering the case where a is a nonresidue: $a^{\frac{p-1}{2}} \equiv -1 \pmod p$. For equations of the form $bx \equiv a \pmod p$, as we have discussed, when we introduced arithmetic modulo a prime number, there is a unique solution b' such that $b \times b' \equiv a \pmod p$. Notice that $b \neq b'$, since, otherwise we would have the case $b^2 \equiv a \pmod p$ and, in consequence, a would be a residue of p, but we are discussing the case that a is a nonresidue. We now apply the same technique as above: we build pairs of bs and b's to simplify the computation of the factorial. For each pair (b, b'), where $b \neq b'$, we have $b \times b' \equiv a \pmod{p}$. Since there is exactly one b' for any b in $1 \dots p-1$, there are $\frac{p-1}{2}$ pairs of (b, b'). For instance:

$1 \times 6 \equiv 6 \pmod{7}$
$2 \times 3 \equiv 6 \pmod{7}$
$4 \times 5 \equiv 6 \pmod{7}$.

When we compute the factorial, we multiply these pairs out, each of which gives a. So, the factorial is $a \times a \times \dots \times a$ and, since there are $(p-1)/2$ such pairs, $a^{\frac{p-1}{2}}$:

$$(p-1)! \equiv a^{\frac{p-a}{2}} \pmod{p}. \tag{5.45}$$

From Wilson's theorem, we know that $(p-1)! \equiv -1 \pmod{p}$. We, therefore, conclude that $a^{\frac{p-a}{2}} \equiv -1 \pmod{p}$. □

Now we consider the other case, *i.e.* that a is a quadratic residue. In this case, we actually have a solution for $x^2 \equiv a \pmod{p}$.

Consider Fermat's little theorem again:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{5.46}$$

We subtract 1 from both sides:

$$a^{p-1} - 1 \equiv 0 \pmod{p} \tag{5.47}$$

and force in our magic formula $\frac{p-1}{2}$ by factoring the left-hand side. What we are doing step-by-step is

$$\begin{gathered} a^{\frac{p-1}{2}} \times a^{\frac{p-1}{2}} = a^{\frac{p-1}{2}+\frac{p-1}{2}} = \\ a^{\frac{(p-1)+(p-1)}{2}} = a^{\frac{2p-2}{2}} = a^{p-1}. \end{gathered}$$

We can reformulate equation 5.47 accordingly:

$$(a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \equiv 0 \pmod{p}. \tag{5.48}$$

Since, to make a product 0, one of the factors must equal 0, $a^{\frac{p-1}{2}}$ must take either the value 1 or -1.

If a is a quadratic residue, then we have some integer x, such that $a \equiv x^2$. We, hence, could write:

$$(x^{2^{\frac{p-1}{2}}} - 1)(x^{2^{\frac{p-1}{2}}} + 1) \equiv 0 \pmod{p}. \tag{5.49}$$

That would mean that, to make the first factor 0, x^2 must be 1 modulo p and, to make the second factor 0, x^2 must be -1 modulo p. We, hence, want either:

$$x^{2^{\frac{p-1}{2}}} \equiv 1 \pmod{p} \tag{5.50}$$

or

$$x^{2^{\frac{p-1}{2}}} \equiv -1 \pmod{p} \tag{5.51}$$

Since x^{a^b} is just x^{ab}, we can simplify to:

$$x^{p-1} \equiv 1 \pmod{p}, \tag{5.52}$$

for equation 5.50 and

$$x^{p-1} \equiv -1 \pmod{p}, \tag{5.53}$$

for equation 5.51. The second result contradicts Fermat as already seen above and, thus, cannot be the case. One may be tempted to say immediately that the second factor cannot be 0, since then we would have $x^2 = -1$, which cannot be true for any integer x. With modular arithmetic, however, this is not true. A counterexample is $2^2 = 4$, which is -1 modulo 5. We therefore need the reference to Fermat's little theorem to actually show that equation 5.51 leads to a contradiction with a proven theorem.

The simplification of equation 5.50, however, just yields the little theorem. We, thus, can derive Euler's criterion directly from Fermat and that proves that, if a is a quadratic residue of p, we always have $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. □

The French mathematician Adrien-Marie Legendre (1752 – 1833) defined a function using Euler's Criterion and a nice notation to express this function known as the *Legendre Symbol*. It can be defined as:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}. \tag{5.54}$$

For $a \in \mathbb{Z}$ and p an odd prime, $\left(\frac{a}{p}\right) \in \{1, -1, 0\}$. More specifically, if a is a residue of p, then $\left(\frac{a}{p}\right)$ is 1, if it is a nonresidue, it is -1, and if $a \equiv 0 \pmod{p}$, it is 0.

The Legrende Symbol has some interesting properties. We will highlight only two of them here. The first is that if $a \equiv b \pmod{p}$, then (trivially)

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right). \tag{5.55}$$

More interesting is multiplicativity:

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right). \tag{5.56}$$

For example, $\left(\frac{3}{11}\right) = 1$ and $\left(\frac{5}{11}\right) = 1$. So $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = 1$ and $\left(\frac{3\times 5=15}{11}\right)$ is 1 as well. $\left(\frac{6}{11}\right) = -1$ and $\left(\frac{3\times 6=18}{11}\right)$ is -1. $\left(\frac{2}{11}\right) = -1$ and $\left(\frac{2\times 6=12}{11}\right) = 1$. A final example with 0: $\left(\frac{22}{11}\right) = 0$ and, with any other number, *e.g.*: $\left(\frac{2\times 22=44}{11}\right) = 0$.

We can implement the Legendre Symbol easily in Haskell as:

```
legendre :: Integer → Integer → Integer
legendre a p = let n = (p − 1) `div` 2
               in case (a ↑ n) `rem` p of
                    0 → 0
                    1 → 1
                    x → x − p
```

For some time now, we are beating around the *law of quadratic reciprocity* and it appears to be high time to finally explain what this law is all about. The law is about two primes, p and q, and claims that if the product

$$\frac{p-1}{2} \times \frac{q-1}{2} = \frac{(p-1)(q-1)}{4}$$

is even, then, if p is a residue of q, q is also a residue of p. Otherwise, if the above product is odd, then, if p is a residue of q, q is nonresidue of p.

This can be formulated much more clearly using the Legendre symbol:

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} \tag{5.57}$$

If $\frac{(p-1)(q-1)}{4}$ is even, then the right-hand side of the equation becomes 1, otherwise it is -1. To become 1, the Legendre Symbols on the left-hand side of the equation must be either both negative or both positive. They are both positive, namely 1, if p is residue of q and q is residue of p. They are both negative, namely -1, if neither p is residue of q nor q of p.

For the right-hand side to become -1, one of the Legendre Symbols must be negative and the other positive. This is the case if either p is residue of q, but q is not residue of p or if q is a residue of p, but p is not a residue of q.

For example, look at the primes 7 and 11. The residues of 7 are $\{0, 1, 2, 4\}$ and, since $11 \equiv 4 \pmod{7}$, 11 is a residue of 7. The residues of 11 are $\{0, 1, 3, 4, 5, 9\}$. 7, hence, is not a residue of 11. Now look at the fraction

$$\frac{(7-1)(11-1)}{4} = \frac{60}{4} = 15,$$

which is odd. Therefore 7 can only be a residue of 11, if 11 is a nonresidue of 7 and vice versa. 11 is a residue of 7, therefore 7 is not a residue of 11.

What about 7 and 29? Look at the magic fraction:

$$\frac{(7-1)(29-1)}{4} = \frac{168}{4} = 42.$$

42 is even, therefore 7 can only be residue of 29 if 29 is a residue of 7. The residues of 29 are: $\{0, 1, 4, 5, 6, 7, 9, 13, 16, 20, 22, 23, 24, 25, 28\}$. Since 7 is included (and 7 hence is a residue of 29), 29 must also be a residue of 7. Since $29 \equiv 1 \pmod{7}$ and 1 is indeed residue of 7, 29 is a residue of 7.

The residues of 5 are $\{0, 1, 4\}$. For 5 and 7, the magic formula is even:

$$\frac{(5-1)(7-1)}{4} = \frac{24}{4} = 6.$$

So, since 5 is a nonresidue of 7, 7 must also be a nonresidue of 5.

The law had already been conjectured by Euler and Legendre, when Gauss finally proved it in the *Disquisitiones*. Gauss called the theorem the *Golden Rule* and, interestingly, the *fundamental theorem of arithmetic* highlighting the value he attached to it. During his life he provided eight different proofs. Many more proofs have been devised since Gauss. According to the *Book*, there were 196 different proofs in the year 2000. We will not go through them here.

5.9. Generators and Subgroups

Let us look at powers of numbers modulo a prime from another angle. In the previous section, we looked at remainders that are squares. Now we look at what happens to remainders, when we raise them to exponents:

$$x^1, x^2, x^3, \dots, x^{p-1} \mod p.$$

What do we expect to happen? We first can predict that, for any number x, there is an exponent k, such that $x^k = 1$. In other words, the set that we create in this way always contains the identity. One case is x^{p-1} for which we know from Fermat's little theorem that it is congruent to 1 for any number x. For instance $3^6 = 1 \pmod 7$. We also know that, if x is a residue, then $x^{\frac{p-1}{2}} = 1$. There, hence, are numbers that result in a smaller set of numbers, since, once $x^k = 1$, the sequence will just repeat with $x^{k+1} = x$, $x^{k+2} = x^2$ and so on.

If x is a nonresidue, we know that $x^{\frac{p-1}{2}} = -1$. Then, $x^{\frac{p-1}{2}} x^{\frac{p-1}{2}} = x^{p-1} = 1$. Since, in the group of any odd prime there are residues and nonresidues, we know for sure that some numbers create the whole group and others do not.

For 7, the powers of 3, for instance, yield the whole group:

$$\begin{aligned} 3^1 &\equiv 3 \pmod 7\\ 3^2 &\equiv 2 \pmod 7\\ 3^3 &\equiv 6 \pmod 7\\ 3^4 &\equiv 4 \pmod 7\\ 3^5 &\equiv 5 \pmod 7\\ 3^6 &\equiv 1 \pmod 7 \end{aligned}$$

The powers of 4, which is a residue of 7, do not:

$$\begin{aligned} 4^1 &\equiv 4 \pmod 7\\ 4^2 &\equiv 2 \pmod 7\\ 4^3 &\equiv 1 \pmod 7. \end{aligned}$$

Second, we observe that we create a set of numbers with certain relations among them:

$$\begin{aligned} a =& x^1\\ b =& x^2\\ c =& x^3\\ & \dots\\ 1 =& x^{p-1} \end{aligned}$$

Any multiplication of two numbers in the set results in another number in the set. Therefore, any power of a number in the set will result in another number in the set. Since $b = aa$ and $c = aaa$, it also holds that $c = ab$. We can go on this way by observing that every number n_i in the set is the result of multiplying the first number in the set a, which is just $x^1 = x$, with its predecessor n_{i-1} or the second number $x^2 = aa$ with n_{i-2} and so on. The set, hence, is closed under multiplication. Furthermore, at some step, n_i becomes 1 and, from any number in the set, we can get to 1 by multiplying another number in the set. This is trivially true for $1 \times x^k = 1$, if k is the number such that $x^k \equiv 1 \pmod{n}$; it is also true for $x^1 x^{k-1} = 1$ and it is in general true for any number $x^i x^{k-i}, 0 \leq i \leq k$. When we have, for instance $k = 3$, then $1 \times aaa = aaa = 1$. $a \times aa = aaa = 1$ and $aaa \times aaa = 1 \times 1 = 1$. In other words, for every element a in the set, there is also its inverse a' in the set, such that $aa' = 1$. That means that the resulting set is again a multiplicative group.

We call x, the number that generates a group G, the *generator* for G. It is often also called a *primitive element* of G. If the group H generated by a number g modulo p is not the whole group G of p, *i.e.* the numbers $\{1, \dots, p-1\}$, then we call H a *proper subgroup* of G. A subgroup H of a group G is a group that contains only numbers that are also in G. G, hence, is a subgroup of G itself. A proper subgroup, H, of a group G is a subgroup, where not all members of G are also in H. A proper subgroup G is therefore smaller than G. The group generated by the generator 3 modulo 7, for instance, is a subgroup of G (it is in fact identical to G). The group generated by the generator 4 modulo 7, too, is a subgroup of G, but it is a proper subgroup, since all elements in this group are also in G, but not all elements in G are in this subgroup. This is the same concept as the subset in set theory.

We can devise a simple function to generate a group, given p, the prime, and g, the generator:

```
generate :: Natural → Natural → [Natural]
generate p g = sort $ nub (map (λa → (g ↑ a) `rem` p) [1 . . p − 1])
```

Note that we *nub* the result to restrict the resulting set to the group itself. For the case where g generates a proper subgroup of the entire group, we otherwise would get repetitions. We also *sort* the groups to get a canonical order, *i.e.* $[1, 2, 4]$ instead of $[4, 2, 1]$.

Let us look at the groups generated by all the numbers $\{1 \dots 6\}$ modulo 7:

generate $7\ 1 = [1]$
generate $7\ 2 = [1, 2, 4]$
generate $7\ 3 = [1, 2, 3, 4, 5, 6]$
generate $7\ 4 = [1, 2, 4]$
generate $7\ 5 = [1, 2, 3, 4, 5, 6]$
generate $7\ 6 = [1, 6]$.

We see 4 different groups. Two of these groups are quite trivial: $g = 1$ generates a group with just 1 element, since $1 \times 1 = 1$; $g = 6$ generates a group with two elements, since $1 \times 6 = 6$ and $6 \times 6 = 1$. These two trivial groups exist for any prime greater 2, since $1 \times 1 = 1$, trivially, holds for any modulus and $(p-1)(p-1) = 1$ holds for any prime modulus. 2 is an exception, because, with 2, we have $1 = p - 1$ and, therefore, 2 has only one trivial group.

The other subgroups modulo 7 are: $\{1, 2, 4\}$ generated by 2 and 4 and the complete group $\{1 \dots p-1\}$ generated by 3 and 5. The size of these groups are 1 and 2 (for the trivial ones) and 3 and 6 for the non-trivial ones. We call the size of a group its *order* and write $|G|$ for the order of group G. The order of the complete prime group is, as we know, $p - 1$. What about the order of the other groups? Is there a pattern too?

To further investigate, we define a function that shows all subgroups of a prime:

```
allGroups :: Natural → [[Natural]]
allGroups p = map (generate p) [2 .. p − 2]
```

Note that we leave out the trivial groups 1 and $\{1, p-1\}$; we know that they exist for any p, so there is not much information added by showing them.

These are the results for *allGroups* 13 (with duplicates already removed):

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
$[1, 3, 9]$
$[1, 3, 4, 9, 10, 12]$
$[1, 5, 8, 12]$

We see again 4 groups. They have the orders 3, 4, 6 and 12. There are also the two trivial groups with order 1 and 2 of course. A striking peculiarity is that for both 7 and 13, all subgroups have orders that divide the order of the main group. For 7, the order of the main group is 6 and the proper subgroups have the orders 1, 2 and 3. For 13, the order of the main group is 12 and the proper subgroups have the orders 1, 2, 3, 4 and 6. We will use the following function to investigate this further:

```
orders :: Natural → [Int]
orders = nub ∘ map length ∘ allGroups
```

The result for *orders* 13 is $[12, 3, 6, 4]$.

We now map this function on the primes; we drop the first two primes, 2 and 3, because they have only the trivial subgroups and start by looking at the first 9 primes starting with 5 calling *map orders* (*take* 9 (*drop* 2 *allprimes*)):

5	7	11	13	17	19	23	29	31
4	3,6	5,10	3,4,6,12	4,8,16	3,6,9,18	11,22	4,7,14,28	3,5,6,10,15,30

The suspicion is confirmed: The orders of the subgroups always divide the order of the prime group. Indeed, this fact is known as *Lagrange's theorem.* It was proven by Joseph

Louis Lagrange, 1736 – 1813, an Italian mathematician who lived and taught in Turin, Berlin and Paris. Lagrange proved another important theorem that we already met: Wilson's theorem.

Lagrange's theorem is a kind of crossroads between different branches of mathematics including group theory, number theory, set theory and algebra. It is as such a quite deep theorem and to appreciate its full meaning, we need much more mathematical machinery than we have available right now. We will come back to Lagrange's theorem and provide a complete proof. Here, we will provide a quite simple proof sufficient for the current context concerning finite multiplicative groups. But even this proof provides surprisingly deep insight into the structure of groups.

Lagrange's theorem states that for any group G and any of its subgroups H: $|H|$ divides $|G|$. We start the proof by considering an arbitrary group modulo a prime. Such a group is generated by a sequence of powers of a: $a^1, a^2, \dots, a^k$, where $a^k = 1$. For sake of explicitness, let us consider a concrete example, say, the group modulo 7, which has order 6. Let the sequence of powers of a, any primitive element of that group, be the sequence:

$$a, b, p-1, b', a', 1.$$

In this group, the placement of b and its inverse b' is arbitrary. The placement of $a = a^1$, $p-1$, a' and 1, however, is on purpose and respects the order in which these numbers necessarily appear, when the numbers reflect the numbers generated by a^1, a^2 and so on. The first number, a^1, trivially is a. The last number a^k is 1. The last but one number is that number in the group that multiplied by a results in 1, *i.e.* the inverse of a. Since a^6, in this example, is 1, a^3 must be its own inverse, *i.e.* a number that multiplied by itself, $a^{3+3} = a^6$, is 1. We know that $p-1$ is the only number, besides 1 itself of course, that is its own inverse.

Since the sequence terminates with $a^6 = 1$, it would repeat with $a^7 = a$, when we continue. We show this in the following table up to exponent $k = 12$:

1	2	3	4	5	6	7	8	9	10	11	12
a	b	p-1	b'	a'	1	a	b	p-1	b'	a'	1

Based on this information, try to imagine the group generated by b. b is aa, so we get (with the headline indicating the exponents of a, not of b!):

1	2	3	4	5	6	7	8	9	10	11	12
	b		b'		1		b		b'		1

Indeed, we know that $b = a^2$ and that $b' = a^4$. Consequently, $bb = aaaa = a^4 = b'$ and, even further, $bbb = a^6 = 1$. We therefore see a group with three members.

The inverse of b would generate another group with three members, but since the exponent of b', which is 4, does not divide the exponent of 1, 6, we need more than one cycle to terminate the group:

1	2	3	4	5	6	7	8	9	10	11	12
			b'				b				1

Continuing this scheme, we can easily imagine the group generated by $p - 1$:

1	2	3	4	5	6	7	8	9	10	11	12
		p-1			1			p-1			1

and that generated by 1:

1	2	3	4	5	6	7	8	9	10	11	12
					1						1

In other words, the periodicity of element 1, which appears every 6 as, and the relation among numbers, that is $b = aa$ and $p - 1 = aaa$ in this example, determine all possible group orders. If $a^i = p - 1$, then $a^{2i} = 1$ determines the order of the group generated by a. If b is member of that group and $b = a^j$, then j must divide either $2i$ or a multiple of that number – a', which is a^5, for instance, would only return to 1 at index 30. Since only every j^{th} element is in the group of b, the group has $\frac{lcm(2i,j)}{j}$ members. For b, we have $j = 2$ and $\frac{lcm(6,2)}{2} = 3$ group members. For b', we have $j = 4$ and $\frac{lcm(6,4)}{4} = 3$ group members. For $p - 1$, we have $j = 3$ and $\frac{lcm(6,3)}{3} = 2$ group members. For 1, we have $j = 6$ and $\frac{lcm(6,6)}{6} = 1$ group member. For a' we have $j = 5$ and $\frac{lcm(6,5)}{5} = 6$ group members. For a, finally, we have $j = 1$ and $\frac{lcm(6,1)}{1} = 6$ group members. The number $\frac{lcm(2i,j)}{j}$, for sure, divides $2i$, since $lcm(2i, j)$ is a multiple of both, $2i$ and j. That is all what we wanted to prove. $\square$

For our case at hand, this proof (even though a bit sloppy using an example) is sufficient. The theorem, however, is not limited to remainders of primes. We will see other examples soon and, indeed, we already saw an example with very similar effects. In the previous section, we discussed composition of permutations and there we saw that a permutation with orbits of different size n and m, need $lcm(n, m)$ applications to come back to the original sequence.

But let us come back to problems of primes. When we look at the table of subgroup orders above, we see some primes, such as 11 and 23, with strikingly fewer subgroups than the primes in their surrounding. 11 and 23, both, have two subgroups, while 13, 19 and 29 all have four subgroups. Is this a pattern or is it just one of the curiosities that arise with small numbers? Here is a list of six more primes generated by *orders* (*take* 6 (*drop* 11 *allprimes*)):

37	41	43	47	53	59
3,4,6,9,12,18,36	4,5,8,10,20,40	3,6,7,14,21,42	23,46	4,13,26,52	29,58

Most primes we see in this sequence have 6 or 7 subgroups and one, 53, has 4. The outliers are 47 and 59 with 2 subgroups each. So, what is special about the primes 11, 23, 47 and 59?

Let us examine their subgroup orders:

11	23	47	59
5,10	11,22	23,46	29,58

All of these primes have the subgroup with order $p - 1$ and a subgroup with order $\frac{p-1}{2}$, *i.e.* the half of the order of the main group. In all cases, the half of the order of the main group is again a prime number: 5, 11, 23 and 29. When the order of the main group has only two prime factors, namely 2 and q, where q is again prime, then, since the order of any subgroup must divide the order of the main group, there cannot be another subgroup besides the trivial ones with order 1 and 2. This fact has huge importance for cryptography, especially the Diffie-Hellman key exchange protocol and the Schnorr signature. These cryptosystems make use of primes of the form $2p+1$ where p is also prime to guarantee that any element chosen but 1 and $p - 1$ is member of a huge subgroup. We will discuss this later in this chapter.

Primes of the form $2p + 1$ (with p prime) are called *safe primes*. The other prime, the p in the safe prime formula, is called *Sophie Germain* prime after the great French mathematician Sophie Germain (1776 – 1831). Germain began to study math with about 13 years of age. Later, when the École Polytechnqiue opened in Paris during the French Revolution, she started to send essays to the teachers there and one of them, again Lagrange, recognised her talent.

Since it was not allowed for women to study at the Polytechnique at that time, Germain used the name of a former student, Antoine-August Leblanc, when she presented her papers. Lagrange, however, was quite excited about the quality of these essays and was eager to get to know this talented student. So, Germain was forced to reveal her identity, when Lagrange invited the person he believed to be Leblanc. Fortunately, Lagrange continued to support Germain and she was able to present important results in mathematics including number theory and won prestigious prizes from the Paris Academy of Sciences.

Germain would also correspond with Gauss, again under the name Leblanc; when the Napoleon army occupied Braunschweig, where Gauss lived at that time, she asked a friend of the family who was actually a general of the French army to see after Gauss' safety during the occupation. On this occasion, Gauss learnt who his French correspondent was and wrote later:

> How can I describe my astonishment and admiration on seeing my esteemed correspondent M leBlanc metamorphosed into this celebrated person… when a woman, because of her sex, our customs and prejudices, encounters infinitely more obstacles than men in familiarising herself with knotty problems, yet overcomes these fetters and penetrates that which is most hidden, she doubtless has the most noble courage, extraordinary talent, and superior genius.

Let us devise a fuction in the honour of Sophie Germain to list the primes that bear her name:

$$\begin{aligned}&\mathit{sophieprimes} :: [\mathit{Natural}]\\&\mathit{sophieprimes} = \mathit{filter}\ (\lambda p \to \mathit{prime}\ (2 * p + 1))\ \mathit{allprimes}\end{aligned}$$

The first 16 Sophie Germain primes, listed with *take* 16 *sophieprimes* are:

2, 3, 5, 11, 23, 29, 41, 53, 83, 89, 113, 131, 173, 179, 191, 233.

Let us confirm that the primes of the form $2p + 1$ that correspond to these Sophie Germain primes all have only two non-trivial groups. We use the function:

$$\begin{aligned}&\mathit{safeprimes} :: [\mathit{Natural}]\\&\mathit{safeprimes} = \mathit{map}\ (\lambda q \to 2 * q + 1)\ \mathit{sophieprimes}\end{aligned}$$

These are 16 safe primes greater than 59 listed with *take* 16 (*dropWhile* ($\leqslant$ 59) *safeprimes*):

83, 107, 167, 179, 227, 263, 347, 359, 383, 467, 479, 503, 563, 587, 719, 839.

The subgroup orders of the first 9 of them are:

83	107	167	179	227	263	347	359	383
41,82	53,106	83,166	89,178	113,226	131,262	173,346	179,358	191,358

and we see that, indeed, all safe primes have exactly two non-trivial groups.

The concept can be extended to primes of the form $p = nq + 1$ where n is even. This way we obtain a p with potentially much more subgroups than just the group of order $\frac{p-1}{2}$, *viz.* one subgroup per prime factor of n. The group we want to use, that of size $\frac{p-1}{n}$, is called the *Schnorr group* of p after the German mathematician and cryptographer Claus Schnorr, who studied the mathematical properties of this group extensively and invented a cryptographic authentication system based on it, the *Schnorr signature*. We will come back to that soon.

5.10. Primality Tests

Primality tests are algorithms that test whether a given number is prime. Until now, we have only one primality test. This test, basically, creates one prime after the other until it finds one that divides the number in question. If the first number found is that number itself, then this number is prime. The test finds an answer after having, in the worst case, examined $\sqrt{n}$ numbers. For small numbers, this is great. For large numbers, however, this may turn out to be very expensive. To be honest, in spite of all its ingenuity, this algorithm is quite shabby. It is just a brute force attack that tries to solve the problem by looking at all primes that stand between us and $\sqrt{n}$.

In the previous sections, we have learnt a lot of facts concerning prime numbers. Perhaps some of those facts may help us to distinguish between composites and primes. The first candidate is Fermat's little theorem, which states that, for any integer a and any prime p:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{5.58}$$

This way, Fermat's theorem provides a criterion for a number being prime (or, more precisely, being not prime) that can be easily tested. Power is still a heavy operation with large numbers, but we need to apply the opertation only once and that is indeed much cheaper than going through millions of primes to test just one number for primality. A test based on Fermat could be implemented like this:

```
fprime :: Natural → Bool
fprime 0 = False
fprime 1 = False
fprime 2 = True
fprime p = (2 ↑ (p − 1)) ‘rem‘ p ≡ 1
```

Since the condition should hold for any integer a, we just choose 2 and check if it, raised to $p-1$, leaves the remainder 1 divided by p. (We do not choose 1, of course, since 1 trivially leaves 1 with any number p.) We could test this new primality test with the old one comparing their results, like this: $[(n, prime\ n, fprime\ n) \mid n \leftarrow [1\,..]]$. The result look like this:

```
(1, False, False)
(2, True, True)
(3, True, True)
(4, False, False)
(5, True, True)
...
```

and without much surprise, we see that *fprime* produces the same results as *prime*. But be careful: Fermat's little theorem claims that all primes adhere to the rule, but it does not make any statement on composites. According to the theorem, composites may or may not leave a remainder of 1 with a^{p-1}. What the criterion establishes is therefore not that p is prime, but that, if the condition is not fulfilled, p is not prime. Is this relevant? Well, let us see what happens, when we continue the listing above:

...

```
(339, False, False)
(340, False, False)
(341, False, True)
...
```

The primality tests disagree on 341! Which one is right? First let us look at the Fermat test:

$$2^{340} \equiv 1 \pmod{341}.$$

According to this test, 341 appears to be prime. So it must not have any factors. However, *trialfact* 341:

```
[11, 31]
```

Apparently, 341 has two factors, 11 and 31, since $11 \times 31 = 341$. So, 341 is definitely not a prime. In fact, there are composites that pass certain primality tests, so called *pseudoprimes*. We could have actually avoided falling into this trap by chosing a different a, for instance:

$$3^{340} \equiv 56 \pmod{341}.$$

However, there are still 98 of 338 numbers in the range $3 \dots 340$ for which the Fermat test would have succeeded. We could repair the Fermat test by making it stronger: we could demand that all numbers $2 \dots 340$ must pass the test, before we accept p being prime. This, however, would make the Fermat test quite expensive – and that it is inexpensive was the main reason we have chosen it in the first place. As an alternative, we could demand that there should be a certain amount of numbers for which the Fermat test should not fail. But even this would not help a lot, since there are numbers where indeed most $a \in \{2 \dots p-1\}$ would pass the test, namely, the *Carmichael numbers*:

561, 1105, 1729, 2465, 2821, 6601, 8911, ...

A Carmichael number n is a composite number such that $a^{n-1} \equiv 1 \pmod{n}$ for every a coprime to n. When Robert Carmichael discovered the first of these numbers in 1910, the notion already existed, but under another name and with another definition. Already in 1899, the German mathematician Alwin Korselt defined numbers n, such that n is squarefree (no prime factor appears more than once in the prime factorisation of that number) and that, for every prime factor p, it holds that $(p-1)|(n-1)$. It turned out that both definitions are equivalent. 561, for example, has the factorisation $\{3, 11, 17\}$. Trivially, $3 - 1 = 2$ divides $561 - 1 = 560$; $11 - 1 = 10$ also divides 560 and, finally, $17 - 1 = 16$ divides 560, since $16 \times 35 = 560$.

Numbers coprime to 561 in the range $1 \dots 560$ are all numbers not multiples of the prime factors 3, 11 or 16. 242 of the 560 numbers $1 \dots 560$ are actually multiples of (at least) one of the prime factors. All other numbers are coprime to 561. In other words, more than half of the numbers will pass the Fermat test. Carmichael numbers are therefore hard to distinguish from primes by means of tests that avoid testing all remainders of n.

There are candidates for much stronger primality tests, however. Wilson's theorem, for instance, provides a criterion that holds for primes only, namely:

$$(p-1)! \equiv -1 \pmod{p}. \tag{5.59}$$

A primality test that would not fall for Carmichael numbers and other pseudoprimes could be based on Wilson's theorem:

```
wprime :: Natural → Bool
wprime 0 = False
wprime 1 = False
wprime 2 = True
wprime n = (fac (n − 1)) `rem` n ≡ (n − 1)
```

And, indeed, *wprime* gives *False* for 341 as it does for any of the Carmichael numbers. Unfortunately, factorial is a very expensive operation rendering *wprime* as ineffective as *prime*.

Another idea is to base primality tests on the observation that only for a prime p it holds that for any $0 < k < p$:

$$\binom{p}{k} \equiv 0 \pmod{p}. \tag{5.60}$$

But, again, we have to test this for all ks. In the case of 561, for instance, 446 of the 560 possible ks fulfil the equation. But computing all binomial coefficients $\binom{p}{1}, \binom{p}{2}, \dots \binom{p}{p-1}$, is not feasible for large numbers.

The next bunch of ideas would use an implication of that fact, such as the *freshman's dream*:

$$(a+b)^p \equiv a^p + b^p \pmod{p} \tag{5.61}$$

Unfortunately, there is a proof (by Ghatage and Scott) that this condition is true exactly if p is prime or ... a Carmichael number. To distinguish primes from Carmichael numbers, it is again necessary to test for all numbers a.

There is finally an efficient test that builds on a variant of freshman's dream, namely the Agrawal-Kayal-Saxena (AKS) test, which exploits the fact that

$$(x-a)^p \equiv (x^p - a) \pmod{p}. \tag{5.62}$$

Agrawal, Kayal and Saxena, a group of Indian mathematicians and computer scientists won the Gödel Prize for their paper "PRIME is in P" where they actually presented the AKS test. The test adopts algebraic methods to avoid computing all possible xs and as to establish that a number is prime. Since we have not looked into algebra yet, we have to postpone the discussion of this algorithm. In practical terms, this is not an issue, since there are algorithms that establish the primality of a number with sufficiently high probability.

It is a very common approach in math and computer science to accept algorithms with bounded error probability if those algorithms are significantly faster or simpler than their deterministic cousins and if an error bound can be given. This bound can then be used to compute the number of repetitions necessary to make the probability of failure small enough to be ignored.

A test able to establish the primality of a number greater than 2 with sufficient probability is the Rabin-Miller test. The mathematical idea is based on Fermat's little theorem, but with some refinement. The reasoning starts with the observation that $p-1$ in a^{p-1} in Fermat's equation is even, since p is prime, as required by the theorem, and $p > 2$, as required by Rabin-Miller. We could therefore represent that number as a series of squares of the form

$$a^{s^{2^{\cdots}}},$$

for some odd integer s. This is of course equivalent to

$$a^{s\times 2\times 2\times \dots}.$$

If p is, say, 5, then the Fermat equation would look like $a^4 \equiv \pmod{5}$, which we could write as $a^{1\times 2\times 2}$, where $s = 1$. For $p = 7$, this would look like $a^{3\times 2}$ and $s = 3$.

Let us look at the "last" square $(a^d)^2$ independent of whether d is even or odd. We know this square must make the equation congruent to 1 modulo p. Let us examine this last exit before $p-1$, a^d, and call this number x. We then have:

$$x^2 \equiv 1 \pmod{p},$$

From here, we can apply the same technique we have already used to prove Wilson's theorem; we first subtract 1 on both sides and we get

$$x^2 - 1 \equiv 0 \pmod{p}.$$

$x^2 - 1$ can be factored into $(x+1)(x-1)$ and we get the congruence

$$(x+1)(x-1) \equiv 0 \pmod{p}.$$

For the product on the left-hand side to become 0, one of its factors must be 0. For the first to be 0, x must equal -1; for the second, x must equal 1. This is the same result we have already obtained, when proving Wilson's theorem.

We substitute a^d back for x and see that the last exit before the last square must be either $a^d \equiv -1 \pmod{p}$ or $a^d \equiv 1 \pmod{p}$. The second case, however, where a^d is 1, can only occur if a^s was 1 or $p-1$ right from the beginning or if somewhere on the way from a^s to a^d, the whole expression becomes $p-1$. We know this for sure from Wilson's theorem: For every number a in the range $1 \dots p-1$, there is an inverse a', such that $aa' \equiv 1 \pmod{p}$ and there are only two numbers for which $a = a'$, namely 1 and $p-1$. Squaring a number that is neither 1 nor $p-1$, therefore, cannot result in 1. There are thus only two ways for a^{2d} to become 1: either a^s was 1 right from the beginning, then squaring will not change anything; or, at some point, a^d is $p-1$ (which may be obtained by squaring two numbers) and then, in the next step, 1. It is impossible, however, that, if a^s was not 1 nor $p-1$, that 1 pops up on the way, without $p-1$ having occurred before.

This is the idea of Rabin-Miller: It finds an odd number s and a number t that tells us how often we have to square a^s to get to a^{p-1}. Then it checks if a^s is either 1 or $p-1$. If not, it checks if any of a^s, a^{2s}, a^{4s}, $\dots$, a^{ts} is $p-1$. If this is not the case, p is composite.

The advantage of this method over the simple Fermat test is that it reduces the set of remainders per number that actually pass. This also reduces the probability of the test to actually go wrong for a specific number. If this probability is significantly less than 50%, we can reach a correct result with high probability by applying the test more than once. But let us postpone the probability reasoning for a short while. First, we will have a look at the implementation of the algorithm.

To start, we need a function that gives us s, the odd number after taking all squares out of $p-1$, and t, the number that tells us how many squares we have actually taken out to reach s. It then holds that $2^t s = p-1$. In the lack of a useful name for that function, we call it *odd2t*:

```
odd2t :: Natural → Natural → (Natural, Natural)
odd2t s t | even s    = odd2t (s `div` 2) (t + 1)
          | otherwise = (s, t)
```

For 16, *odd2t* would give $(1, 4)$, since 16 is a power of 2, *i.e.* divided subsequently by 2, it will reach 1. One has to multiply 1 4 times by 2 to get 16 back: $1 \times 2^4 = 16$. For 18, *odd2t*, accordingly, would yield $(9, 1)$, since $9 \times 2^1 = 18$.

The next function is the primality test itself:

```
rmPrime :: Natural → Natural → Natural → Natural → Bool
rmPrime p a s t = case (a ↑ s) `rem` p of
                    1 → True
                    v → if v ≡ p − 1 then True
                                     else go t v
  where go 1 _ = False
        go t v = case (v ↑ 2) `rem` p of
                   1 → False
                   v′ → if v′ ≡ p − 1 then True
                                      else go (t − 1) v′
```

The function receives four arguments: The number to test for primality, the test candidate a, also called a witness for the primality of p, and s and t obtained from *odd2t*. The function raises a to the power of s. If the result is 1 or $p - 1$ modulo p, p has already passed the test. Otherwise, we loop through *go*. *go* receives two argument t and v (which initially is a^s). If t is 1, we have exhausted all the squares in $p - 1$ without having seen $p - 1$. The test has failed. Otherwise, we create the next square modulo p. If this square is 1, something is wrong: we know that v was neither 1 nor $p - 1$, so squaring it cannot result in 1, if p is prime, because only 1 and $p - 1$ are their own inverses. Otherwise, if the result is $p - 1$, we are done. Further squaring will yield 1 and all conditions for p being a prime are fulfilled. Otherwise, we continue with the next square, reducing the square counter t by 1.

The function that brings these bits together needs randomness to choose as. To this end, we use the function *randomNatural* defined in the previous chapter:

```
rabinMiller :: Natural → Natural → IO Bool
rabinMiller _ 0 = return False
rabinMiller _ 1 = return False
rabinMiller _ 2 = return True
rabinMiller k p | even p    = return False
                | otherwise = let (s, t) = odd2t (p − 1) 0 in go k s t
  where go 0 _ _ = return True
        go i s t = do a ← randomNatural (2, p − 1)
                      if rmPrime p a s t then go (i − 1) s t
                                         else return False
```

The function receives two arguments: k and p. p is the number under test. k tells the function how often it has to repeat the test until the expected probability is reached. If k is exhausted, *i.e.* $k = 0$, we return *True* (in *go*) and p has passed the complete test.

At the beginning, we take care of some trivial cases, such as 0 and 1, which are never prime, and 2, which actually is prime. With the exception 2, no even number is prime. Then we start the hard work: we first find s and t using *odd2t*; then we enter *go*. We generate an a from the range $2 \dots p - 1$, using *randomNatural*. Then we apply the test.

If the test fails, we immediately return False. If the test passes, we repeat until $i = 0$.

To reason about the probability for the test to fail, let us look at some examples. The following simple function can be applied to a number to show the results of the Fermat test. It returns a list of tuples where the first element is one of the numbers $2 \dots n-1$ and the second is this number raised to $n-1 \bmod n$:

```
rest :: Natural → [(Natural, Natural)]
rest n = zip rs $ map (λa → (a ↑ (n − 1)) `rem` n) rs
  where rs = [2 . . n]
```

rest 9, for instance, yields:

$[(2,4),(3,0),(4,7),(5,7),(6,0),(7,4),(8,1)]$

We see that, for most of the numbers, the Fermat test would fail. For 8, however, it would pass, since $8^8 \equiv 1 \pmod 9$. 8 is therefore a *liar* concerning the primality (or, more precisely, for the compositeness) of 9. Unfortunately, Rabin-Miller would not help us in this case, since *odd2t* 8 0 = (1, 3); 8^1, however, is $n-1$ and the test would immediately pass. 8, hence, is a *strong liar* for 9.

Let us look at another example: 15. *odd2t* for 15 gives $(7, 1)$, since 14 ‘*div*‘ 2 is 7, which is odd. *rest* 15 yields:

$[(2,4),(3,9),(4,1),(5,10),(6,6),(7,4),(8,4),(9,6),(10,10),(11,1),(12,9),(13,4),(14,1)]$.

There are several liars: 4, 11 and 14. 14, again is a strong liar, since $14^7 \bmod 15 = 14$, which is $n-1$. 11 and 4, however, are ruled out by Rabin-Miller: $11^7 \bmod 15 = 11$, which squared would never be 1, if 15 were prime; $4^7 \bmod 15 = 4$, which squared, again, would not result in 1, if 15 were prime.

Let us devise a function that counts the occurrences of (Fermat) liars and strong liars for any given composite n. The first function is called *liars* and quite simple:

```
liars :: Natural → Int
liars = length ∘ filter (≡ 1) ∘ map snd ∘ rest
```

That is we start with *rest*, ignore the first element of each tuple, filter the 1s and count the elements of the resulting list. (The return type of the function is *Int*, rather than *Natural*, because we use *length*, which returns an *Int* anyway.)

The strong liar function is a bit more tricky:

```
strongLiars :: Natural → Int
strongLiars n | even n     = 0
              | otherwise =
  let (s, t) = odd2t (n − 1) 0
      sl     = foldr (λa l → detector l a s t) [] [2 . . n − 1]
  in  length sl
  where detector l a s t | rmPrime n a s t = a : l
                         | otherwise       = l
```

For finding strong liars, we implement a part of the Rabin-Miller test and, therefore, we ignore even numbers (just yielding 0). For even numbers, the (s,t) values would not make any sense, since $n-1$ is odd! Then, we apply the *rmPrime* test we implemented for Rabin-Miller to all remainders, adding those that pass to the result list. Finally, we just yield the length of that list.

If we apply *liars* to a prime number p, all witnesses $2 \dots p-1$ are counted as liars, *e.g.* *liars* 11: 9. The same is true for *strongLiars*, since the fact that all witnesses are strong liars could be a definition of primalitiy. Thus, *strongLiars* 11: 9.

Applied to 9, *liars* and *strongLiars* yield 1. Applied to 15, *liars* yields 3; *strongLiars*, however, yields only 1. This is in-line with our investigation above. Here are some more examples for numbers between 21 and 75:

21	25	27	33	35	39	45	49	51	55	57	63	65	69	75
3,1	3,3	1,1	3,1	3,1	3,1	7,1	5,5	3,1	3,1	3,1	3,1	15,5	3,1	3,1

We see a quite colourful picture. Many numbers have 3 liars and 1 strong liar; for some numbers, there is no difference in liars and strong liars, for instance 25 and 27, both have the same numbers of liars and strong liars, namely 3 and 1. Other numbers, *e.g.* 45, show a strong reduction in going from liars to strong liars. There are some peaks, *e.g.* 65 has 15 liars and 5 strong liars, much more liars than most other numbers. If we continue up to 99, the greatest number we will see is $(35,17)$ for 91. The nasty number 341 has 99 liars and 49 strong liars. Finally, here are the dreadful Carmichael numbers:

561	1105	1729	2465	2821	6601	8911
319,9	767,29	1295,161	1791,69	2159,269	5279,329	7127,1781

For many Carmichael numbers, the reduction of liars is significant – for some, the reduction is about factor 10 – 30. There are some exceptions with reduction of a factor of "only" 6 like 8911. In general, the number of strong liars is very low compared to n, the prime candidate. It can be shown in fact that the number of strong liars for an odd composite n is at most $\frac{n}{4}$. (This has actually been shown with contributions, among others, by the legendary Paul Erdős, 1913 – 1996.) The arguments, however, are much beyond our scope.

The ratio $\frac{n}{4}$ implies that the probability of hitting a strong liar, when performing *rmPrime* on a randomly chosen witness for n, is $\frac{1}{4}$. In other words, one has to try four times in average to get a strong liar by chance. When we repeat the test several times, we reduce the probability that **all** witnesses we have used are strong liars. It is important to notice that the test yields *False* immediately, when we find a witness for compositeness. We continue only if all tests so far have been witnesses for primality.

The probability is therefore computed as $\frac{1}{4^k}$, where k is the number of repetitions. The probability to obtain two liars in two applications is $\frac{1}{4^2} = \frac{1}{16}$. We, hence, would have to call a Rabin-Miller Test with two repetitions 16 times in average to test only on strong liars once. With four repetitions, the denominator is $4^4 = 64$, with eight, it is 65 536, with sixteen, it is 4 294 967 296 and so on. A reasonable value for k to defend against malicious attacks given, for example, in *Cryptographic Engineering* is $k = 64$. In average, one has a chance of 1 out of 4^{64} or 340 282 366 920 938 463 463 374 607 431 768 211 456 to hit only strong liars in testing a number for primality. That, indeed, appears to be reasonable. A final version of the Rabin-Miller Test could then look like this:

```
rmptest :: Natural → IO Bool
rmptest 0 = return False
rmptest 1 = return False
rmptest 2 = return True
rmptest n | even n    = return False
          | n < 64    = return (wprime (fromIntegral n))
          | otherwise = rabinMiller 64 n
```

5.11. Primes in Cryptography

To say this right at the beginning: this is not an introduction to cryptography! We will go through some examples of how primes are used in cryptography, in particular the Diffie-Hellman key exchange protocol and RSA. But we will not discuss pitfalls, common errors or other issues you have to take care of when implementing cryptosystems. If you want to learn about cryptography, you definitely have to study the literature on the topic. Cryptography poses very hard engineering challenges and we are far from adressing them here. Again: this is not an introduction to cryptography! We do not even use a big number library that would be able to cope with numbers with hundreds or thousands of digits. When you use the algorithms presented here (like the Rabin-Miller test) on numbers of that size, you will probably have to wait minutes or even hours for results. Raising a number of thousands of digits to another number of thousands of digits requests special handling. You cannot do that with our *Natural* data type or even the Haskell *Integer* data type. So, once again: this is not an introduction to cryptography.

Primes come into play in cryptography, typically, in very special applications. Usually, to encrypt and decrypt a message, secret keys are used that are known to both sides of the communication, traditionally called Alice and Bob in the literature. The algorithms to convert the plain message into the *cyphertext* have in most cases nothing to do with primes, but are rather combinations of basic operations like XOR and *bit shuffle* involving the original message and the key material.

The weakest link in this kind of cryptography is the key itself. All parties that take part in the secure communication must know the key and, in consequence, the key must be shared among them in a secure way. How to share the key safely is indeed a major challenge. For the German submarines in World War II, just to name a popular example, the loss of a codebook was at least as challenging as the cryptoanalysis by British specialists at Bletchley Park. A new codebook could hardly be distributed among all submarines on the ocean in due time.

The challenge is an instance of the bootstrapping problem: we have to start a process, namely secure communication, without having the means to run this process, namely secret keys distributed among all parties. One solution is to use publicly available information to convert a plain message into a cyphertext by means of a *one-way function*, *i.e.* a function that is easy to calculate, but difficult to revert. Here is where primes come in.

A one-way function f, is a function that computes a result from a given input, such as $f(x) = y$, for which no inverse f' is known, such that $f'(y) = x$ or $f'(f(x)) = x$. Plain multiplication, for instance, is not a good one-way function, since with the result y known, we simply can revert the effect by division. If we have a function $f(x) = ax$, and a cyphertext $f(x) = y$. We can reconstruct x, simply, by $f'(y) = y/a$. Multiplication would be a good one-way function, however, if both a and x were unknown. The inverse, would then be factoring – and factoring of large numbers is indeed a hard problem.

Before we have a closer look at concrete examples, we have to come back to a question we have already discussed (but without satisfying results), namely how to generate huge primes. We have seen Mersenne primes and Fermat primes, but little is known about this kind of numbers and in particular, no recipe is known how to find new ones. Practial algorithms are in fact much simpler in terms of mathematical ideas. Prime generators usually generate a random number in a given range, test whether it is prime and, if it is, return this number or, if it is not, try again.

A reasonable prime generator for natural numbers, using once again the random number generator *randomNatural*, could look like this:

```
generatePrime :: Natural → IO Natural
generatePrime k = do n ← randomNatural (2 ↑ (k − 1), 2 ↑ k − 1)
                     t ← rmptest n
                     if t then return n
                          else generatePrime k
```

As you can see, this is just a trial-and-error approach using the Rabin-Miller test to check whether a given number is prime. Is there not a risk of running a lot of time depending on the range we have chosen? Indeed, it can take a lot of time, before we actually find a prime. However, we know some things about the distribution of primes, so that we are guaranteed to find a prime within a reasonably chosen range. We will discuss some of the relevant aspects concerning the distribution of primes in the next section. For the moment, it may suffice to mention that a range like $k \dots k+c$, where c is some constant smaller than k, is obviously not sufficient. The range above, however, is chosen in terms of powers of 2 and, assuming that k is at least 10, we are guaranteed to find some prime numbers in such a range like, for $k = 10$, $512 \dots 1023$. It is quite common, by the way, to give the size of the prime wanted in bits, *i.e.* in terms of the number of digits in binary representation, rather than in decimal representation.

The first example of prime-based key exchange is the Diffie-Hellman protocol developed by Whitfield Diffie, Martin Hellman and Ralph Merkle in the 70ies. There was another group of scientists that developed a similar algorithm shortly before Diffie and Hellman published their results. Those scientists, unfortunately, were working for the British secret service, GCHQ, at the time and were not allowed to publish their results. They, thus, escaped eternity.

The Diffie-Hellman protocol is very simple and elegant, but has some pitfalls that must be addressed. Its main purpose is to exchange a secret key between Alice and Bob without Eve, another fictional character famous in cryptography literature, getting to know that key by eavesdropping. Before the Diffie-Hellman protocol actually starts, Alice and Bob have agreed somehow on a public key that may be known to anyone including Eve. This public key may be assigned to either Alice or Bob and may be registered in a kind of phone book or it may be agreed upon in the *handshaking* phase of the protocol.

The public key consists of two parts: a prime number p and a generator g. When Alice initiates the protocol, she chooses a number x from the range $2 \dots p-2$, computes $g^x \bmod p$ and sends this number to Bob. Bob, in his turn, chooses a number y from the same range, computes $g^y \bmod p$ and sends the result back to Alice. Evil Eve knows p and g and may see g^x and g^y. But none of these values is actually the key. The key, instead, is g^{xy}. At the end of the protocol, this number is known to Alice and Bob. Each of them just has to raise the number he or she receives from the other by his or her own number. So Alice chooses x and sends g^x to Bob. Bob chooses y and sends g^y. Alice computes the key as $k = g^{y^x}$, *i.e.* she raises the number she receives from Bob to her own number; Bob computes the key, accordingly, as g^{x^y}, *i.e.* he raises the number he receives from Alice to his own number. That is all.

For a simple example, let us assume the public key is $p = 11$ and $g = 6$. Now, Alice chooses a random number from the range $2 \dots 9$, say, 4, and Bob likewise, say, 3. Alice computes $6^4 \bmod 11 = 9$ and sends 9 to Bob. Bob computes $6^3 \bmod 11 = 7$ and sends 7 to Alice. Alice computes the key as $k = 7^4 \bmod 11 = 3$ and Bob computes it as $k = 9^3 \bmod 11 = 3$. The result is the same for both, of course, because, eventually, both have performed the same operation: g^{xy}.

The security is based on the difficulty to solve the equation

$$k = g^{xy}, \tag{5.63}$$

where g, g^x and g^y are known. This is an instance of the *discrete logarithm* problem, the logarithm in a finite field. If k, g and xy were ordinary real numbers, we could solve the equation in three steps: $x = log_g(g^x)$, $y = log_g(g^y)$ and, finally, $k = g^{xy}$. For the discrete logarithm, however, no efficient solution is known today.

In a trivial examples like the one we used above, Eve can simply try out all possible combinations. In real cryptography applications, we therefore have to use very large primes. But the security of the algorithm also depends on the order of g. For instance, if we want a security that corresponds to 1000 bits (a number with more than 300 digits in decimal representation), then the order of g should be much more than some hundreds or thousands or even millions of numbers generated by that g. To choose a proper g is therefore essential for the strength of the protocol. The question now is: how to choose a proper g?

If you have carefully read the previous sections, you already know the answer: we must use a g from the Schnorr group of a safe prime. Indeed, knowing the order of g is again a hard problem. We have to solve the equation $g^k = 1 \mod p$, which is again an instance of the discrete logarithm. We can circumvent this problem of finding an appropriate group by finding an appropriate prime. With a safe prime, the selection of an appropriate g is indeed simple. We just have to avoid one of the trivial groups, that is we have to avoid 1 and $p - 1$, then we are guaranteed that g is in the group of the Sophie Germain prime q or in the larger group of the safe prime $2q + 1$.

Sophie Germain primes, hence, give us a means to reduce the difficult problem of finding a proper g to the much simpler problem of selecting a proper prime:

```
safePrime :: Natural → IO Natural
safePrime k = generatePrime k ≫= λq → let p = 2 ∗ q + 1 in do
  t ← rmptest p
  if t then return p
       else safePrime k
```

This function generates a *safe prime*, *i.e.* a prime of the form $2q + 1$, where q is a Sophie Germain prime. It just generates a random prime q, doubles it and adds 1 and checks whether the resulting number p is again prime and, if not, repeats the process. From the group of this prime, we can then take a random g, $g \neq 1, g \neq p - 1$, and this g belongs in the worst case to the group q of order $\frac{p-1}{2}$. Since q was generated according to security level k, the group of q is exactly what we need and our main issue is solved.

There is still a problem, though. From the communication between Alice and Bob, Eve sees g^x and g^y. If she has read the section on quadratic residues, she can determine whether g is a square (modulo p) using the Legendre symbol. If g is a nonresidue, then she has an attack: she can repeat the test on the numbers she sees, namely, g^x and g^y.

If a number of the form g^z is a nonresidue, where g as well is a nonresidue, then z is odd, otherwise, z is even. This is because even exponents are just repeated squares. So, if z is even, g^z is a residue and, otherwise, it is not.

We should avoid this leakage. Even though Eve does not learn the whole number, she gets an information she is not entitled to have. The leakage effectively reduces the security level by the factor 2, since Eve learns the least significant digit of the number in binary representation: odd numbers in binary representation end on 1, even numbers on 0. We can avoid this problem simply by chosing a residue right from the beginning. The function to generate an appropriate g could then look like this:

```
generator :: Natural → IO Natural
generator p = do a ← randomNatural (2, p − 2)
                 let g = (a ↑ 2) ‘rem‘ p
                 if g ≡ p − 1 then generator p
                              else return g
```

We select an a from the range $2 \dots p-2$ and square it. The number, hence, is guaranteed to be a residue eliminating the problem discussed above. We test if $g = p - 1$, which is forbidden, since it is in a trivial group. a^2, as you know, can become $p - 1$, if $p \equiv 1 \pmod 4$. You may argue that this is not the case, when p is a safe prime, because, if $\frac{p-1}{2}$ is prime, then $p - 1$ cannot be a multiple of 4. It is a good practice, however, to keep different components of the security infrastructure independent of each other. In spite of the fact that we usually use *generator* with safe primes, it could happen that someone uses it with another kind of prime (for example a prime of the form $nq + 1$). This test simply avoids that anything bad happens under such circumstances.

Note that we really do not need to check for $g = 1$, since a^2, with a chosen from the range $2 \dots p - 2$, excluding both 1 and $p - 1$, can never be 1.

The next function initialises the protocol. It is assumed that p and g are known already to all involved parties when it is called:

```
initProtocol :: Chan Natural → Chan Natural →
                Natural → Natural → IO Natural
initProtocol inch outch p g = do
  x  ← randomNatural (2, p − 2)
  let gx = (g ↑ x) ‘rem‘ p
  writeChan outch gx
  gy ← readChan inch
  unless (checkG p gy) $ error ("suspicious value: " ++ show gy)
  return ((gy ↑ x) ‘rem‘ p)
```

The function receives four arguments: An input channel and an output channel and p and g. It starts by generating a random x from the range $2 \dots p - 2$. It then computes $g^x \bmod p$ and sends the result through the outgoing channel. Then it waits for an answer through the incoming channel. When a response is received, the value is checked by *checkG*, at which we will look in an instant. If the value is accepted, the function returns this value raised to x modulo p. This is the secret key.

It is actually necessary to check incoming values, since Eve may have intercepted the communication and may have sent a value that is not in the Schnorr group. To protect against this *man-in-the-middle* attack, both sides apply the following tests:

```
checkG :: Natural → Natural → Bool
checkG p x = x ≢ 1 ∧
             x < p ∧
             legendre (fromIntegral x)
                      (fromIntegral p) ≡ 1
```

For any value x received through the channel, it must hold that $x \neq 1$ (because 1 is in the wrong group), it must also hold that $x < p$, *i.e.* it must be a value modulo p, and x must be a residue of p. This is because we have chosen g to be a square and any square raised to some power is still a residue of p. So, if one of these conditions does not hold, something is wrong and we immediately abort the protocol.

Now, we look at the other side of the communication:

```
acceptProtocol :: Chan Natural → Chan Natural →
                  Natural → Natural → IO Natural
acceptProtocol inch outch p g = do
  gx ← readChan inch
  unless (checkG p gx) $ error ("suspicious value: " ++ show gx)
  y  ← randomNatural (2, p − 2)
  let gy = (g ↑ y) ‘rem‘ p
  writeChan outch gy
  return ((gx ↑ y) ‘rem‘ p)
```

The function starts by waiting on input. When it receives some input, it checks it using *checkG*. If the value is accepted, the function generates a random y from the range $2 \dots p-2$, computes $g^y \bmod p$ and sends it back; finally the key is returned.

Now, the protocol has terminated, both sides, Alice and Bob, know the key and they can start to use this key to encrypt the messages exchanged on the channel. Since, as mentioned several times, this is not an introduction to cryptography, we have omitted a lot of details. One example is a robust defence against *denial-of-service* attacks. In the code above, we wait for input forever. This is never a good idea. An attacker could initiate one protocol after the other without terminating any of them. The server waiting for requests would quickly run out of resources.

A cryptosystem with somewhat different purposes than Diffie-Hellman is RSA, named after its inventors Ronald Rivest, Adi Shamir and Leonard Adleman and published in 1978. Similar to Diffie-Hellman, RSA aims to provide an encryption system for key exchange, but, beyond encryption, RSA is also designed as an authentication system based on electronic signatures. The latter implies that the public key is assigned to a person. Furthermore, an infrastructure is needed that inspires the confidence that a given public key is really the key of the person one believes one is communicating with. There are many ways to create such an infrastructure. It may be community-based, for example, so that people one trusts guarantee for people they trust; another way is that trusted organisations provide phone books where signatures can be looked up. All possible implementations depend at some point on trust. In fact, trust is just the other side of the security coin: without trust, there is no security.

When Alice uses RSA to send a secret message to Bob, she uses Bob's public key to encrypt the message, and Bob, later, uses his private key to decrypt the message. The design of RSA guarantees that it is extremely difficult to decrypt the cyphertext without knowledge of Bob's private key. If Alice, additionally, wants to sign the message to make sure that Eve cannot exchange her messages by other ones, she uses her own private key to create a signature on the message (usually by first creating a *hash* of that message). Bob can then assure himself that the message was really sent by Alice by verifying the signature using Alice's public key. The role of public and private key, hence, is swapped in encryption and signing. Encryption is done with the addressee's pupblic key and undone by the addresse's private key; signing is done with the sender's private key and approved with the sender's public key.

Just as Diffie-Hellman, RSA is based on modular arithmetic – but with a composite modulus. With a composite modulus, some care must be taken, of course, since it does behave as a prime only with respect to those numbers that are coprime to it. The modulus n is created by multiplying two large primes, p and q. This guarantees that n behaves like an "ordinary" prime with respect to most numbers in the range $1 \dots n-1$, *viz.* all numbers that are not multiples of p and q.

We also need a number t, such that for numbers a coprime to n, it holds that $a^t \equiv 1 \pmod{n}$. For a prime number p, as we know from Fermat's little theorem, this t would be $p-1$. For the prime factors of n, hence, $p-1$ and $q-1$ would do the trick. But any multiple of $p-1$ and $q-1$ would do the trick as well, including of course $(p-1)(q-1)$ or $\text{lcm}(p-1, q-1)$, the least common multiple of $p-1$ and $q-1$.

Now, consider Fermat's theorem written like this:

$$a^p \equiv a \pmod{p} \tag{5.64}$$

and try to solve the following congruence system:

$$\begin{aligned} x &\equiv a \pmod{p} \\ x &\equiv a \pmod{q}. \end{aligned}$$

An obvious solution to this system, according to the Chinese Remainder theorem, is the number x that is congruent to a modulo pq. Let us write x as a^t, where $t = \operatorname{lcm}(p-1, q-1)$:

$$\begin{aligned} a^t &\equiv a \pmod{p} \\ a^t &\equiv a \pmod{q}. \end{aligned}$$

An obvious solution to this system, still according to the Chinese Remainder theorem, is $a^t \equiv a \pmod{pq}$. Let us look at an example: $p = 7$, $q = 11$ and $n = pq = 77$. The lcm of $p - 1 = 6$ and $q - 1 = 10$ is 30. Therefore, for any number a: $a^{30} \equiv 1 \pmod{7}$ and $a^{30} \equiv 1 \pmod{11}$, but also: $a^{31} \equiv a \pmod{7}$ and $a^{31} \equiv a \pmod{11}$ and, this is important, $a^{30} \equiv 1 \pmod{77}$ and $a^{31} \equiv a \pmod{77}$. For instance $a = 3$: $3^{30} \equiv 1 \pmod{7}$, $3^{31} \equiv 3 \pmod{7}$ and $3^{30} \equiv 1 \pmod{11}$ and $3^{31} \equiv 3 \pmod{11}$. But also: $3^{30} \equiv 1 \pmod{77}$ and $3^{31} \equiv 3 \pmod{77}$.

Note that this is nothing new. We just applied the Chinese Remainder theorem to a quite trivial case. However, from this trivial case (and with some help from Euler as we will see later), an important theorem follows, namely *Carmichael's theorem* that can be stated in the scope of our problem here as: the least number t fulfilling the congruence $a^t \equiv 1 \pmod{n}$ for any number a coprime to n is the lcm of $p_1 - 1, p_2 - 1, \dots p_s - 1$, where $p_1 \dots p_s$ are the prime factors of n. Since our number n has only two prime factors, namely p and q, our t is $\operatorname{lcm}(p-1, q-1)$. So, finally, this fellow Carmichael is not only bugging us with crazy numbers, but he actually lends a hand to solve a problem once in a while! Thanks Robert!

The importance for the RSA system is related to the fact that we need a public number to compute the cyphertext and a private number to reconstruct the original message. The method to compute the cyphertext is exponentiation. For this purpose, we need an exponent called e. The number e is chosen such that it is a small odd number, $1 < e < t$ and e does not divide t nor n. Now we find the inverse e' of e mod t, such that $ee' \equiv 1 \pmod{t}$.

When Alice encrypts a message m, she computes $c = m^e \bmod n$. To decrypt the cyphertext c, Bob computes $c^{e'}$, which is $m = m^{e^{e'}} = m^{ee'} \bmod n$. Modulo t $m^{ee'}$ would just be $m^1 = m$, since $ee' \equiv 1 \pmod{t}$. Modulo n, this number is some multiple of t plus 1: $ee' \equiv kt + 1 \pmod{n}$. We, hence, have $m^{kt+1} = m^{kt} \times m = m^{t^k} \times m$. But since $m^t \equiv 1 \pmod{n}$, due to the Carmichael theorem, we have $1^k \times m = 1 \times m = m$. Again, Mr Carmichael, thank you!

To resume what we need for RSA: We have a public key (n, e) and a private key (p, q, t, e'), where $n = pq$, $t = \operatorname{lcm}((p-1), (q-1))$ and $ee' \equiv 1 \pmod{t}$. It is essential that all components of the private key remain secret. Any component that leaks out helps Eve reveal the entire private key. The core of the secret is e', since it can be used directly to decrypt encrypted messages and to sign messages in the name of its owner. With t revealed, e' can be simply computed with the extended gcd algorithm; with one of p or q revealed, the respective other factor of n can be simply computed by $q = n/p$ or $p = n/q$. With p and q both known, however, t can be computed by means of the lcm. This boils down to the fact that the security of RSA depends on the difficulty of the discrete logarithm and the factoring of large numbers.

Since, in any concrete implementation of RSA, we have to refer to the components of the keys quite often, let us define data types to encapsulate the public and the private information:

```
type PublicK = (Natural, Natural)
type PrivateK = (Natural, Natural, Natural, Natural)

pubN, pubE :: PublicK → Natural
pubN = fst
pubE = snd

privP, privQ, privT, privD :: PrivateK → Natural
privP (p, _, _, _) = p
privQ (_, q, _, _) = q
privT (_, _, t, _) = t
privD (_, _, _, d) = d
```

This is just two type synonyms for private and public key and a set of accessor functions. Note that we call the inverse of e in the private key d as it is often referred to in the literature.

As for Diffie-Hellman, chosing good values for public and private key is an essential part of the system. The following function is a reasonable key generator:

```
generateKeys :: Natural → IO (PublicK, PrivateK)
generateKeys k = do p      ← genPrime 1
                    q      ← genPrime p
                    let t = lcm (p − 1) (q − 1)
                    pair   ← findED 1000 t
                    case pair of
                      Nothing     → generateKeys k
                      Just (e, d) → if  e ≡ d ∨
                                        e ≡ p ∨ e ≡ q ∨
                                        d ≡ p ∨ d ≡ q
                                    then generateKeys k
                                    else let pub = (p ∗ q, e)
                                             priv = (p, q, t, d)
                                         in  return (pub, priv)
  where genPrime p = do q ← generatePrime (k ‘div‘ 2)
                        if p ≡ q then genPrime p
                                 else  return q
```

We start by generating two primes using *genPrime*. *genPrime* incorporates a test to ensure that we do not accidently choose the same prime twice. When we call *genPrime* for p, we pass a number that is certainly not equal to the generated prime, namely one. In the second call to generate q, we pass p. Then we create t as $\mathrm{lcm}(p-1, q-1)$ and then we find the pair (e, d) using *findED*, at which we will look next. *findED* returns a *Maybe* value. If the result is *Nothing*, we start all over again. Otherwise, we ensure that e and d differ and that e and d differ from p and q. This is to ensure that we do not accidently publish one of the secrets, namely e or one of the prime factors of n. Finally, we create the public key as (pq, e) and the private key as (p, q, t, d).

The most intriguing part of the key generator is finding the pair (e, d). Since t is not a prime number, we are not guaranteed to find an inverse for any $e < t$. So, we may need several tries to find an e. But it might even be that there is no such pair at all for t. Since t depends on the primes p and q, in such a case, we have to start from the beginning. As an example, consider the primes $p = 5$ and $q = 7$; t in this case is 12. If we try all combinations of the numbers $2 \dots 10$, we see that the only pairs of numbers whose product is 1 are 5×5 and 7×7. In this case: $e = d$, an option that any attacker will probably try first and should therefore be avoided. Here is an implementation of *findED*:

```
findED :: Natural → Natural → IO (Maybe (Natural, Natural))
findED 0 _ = return Nothing
findED i t  = randomNatural (7, t − 2) ≫= λx →
  let z | even x        = x − 1
        | otherwise     = x
      (e, d)            = tryXgcd z t
  in if (e ∗ d) ‘rem‘ t ≡ 1 then if e > t ‘div‘ 2 then return (Just (d, e))
                                                  else return (Just (e, d))
                           else findED (i − 1) t
  where tryXgcd a t = case nxgcd a t of
                        (1, k) → (a, k)
                        _      → (0, 0)
```

The function takes two arguments. The first is a counter that, when expired, indicates that we give up the search with the given t. As you can see above in *generateKeys*, the counter is defined as 1000. That is just some randomly chosen value – there may be better ones. We could try to invent some ratio for t such as half of the odd numbers smaller than t. But t is a very large number. Any ratio would force us to test single numbers for hours if we are unlucky.

We then choose a random number from the range $7 \dots t-2$ as a candidate for e. We do not want $e = t-1$, since the inverse in this case is likely to be e itself. We start with 7, since we want to avoid very small values for e. We also want e to be odd. Otherwise, it will not be coprime to t, which is even.

We then compute (e, d) by means of *tryXgcd*. This function computes the *gcd* and the inverse by means of *nxgcd*, which we defined in the section on modular arithmetic. If the *gcd* is not 1, then we are unlucky, since e and t must be coprime. In this case, we return $(0,0)$, a pair that certainly will not pass the test $ed \equiv 1 \pmod{t}$ and, this way, we cause the next try of *findED*. Otherwise, we return the pair (a,k), e and its inverse modulo t.

If we have found a suitable pair in *findED*, we return this pair either as (e,d) or as (d,e), if $e > t`div`2$. The reasoning is that encryption will be more efficient with a small e. On the other hand, we do not want to impose any explicit property on d (such as $d > e$), since that would be a hint that reduces the security level.

The key generator is usually not used, when we establish a secure communication. Instead, the key pair is considered to be stable as long as it has not been compromised by loss or by an attack on the server where key pairs are stored. Some stability is necessary for the authentication part of RSA. If public keys changed frequently, it would be more difficult to be sure that a given key really belongs to the person one thinks it belongs to.

Once the keys are available and the public key has been published, Alice can encrypt a message to Bob using the encryption function:

$$\begin{aligned}&\mathit{encrypt} :: \mathit{PublicK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{encrypt}\ \mathit{pub}\ m = (m \uparrow (\mathit{pubE}\ \mathit{pub}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

Alice uses this function with Bob's public key and a message m represented as a (large) integer value. The function raises the message m to e and takes the result modular n. Bob can decrypt the cyphertext using his private key:

$$\begin{aligned}&\mathit{decrypt} :: \mathit{PublicK} \to \mathit{PrivateK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{decrypt}\ \mathit{pub}\ \mathit{priv}\ c = (c \uparrow (\mathit{privD}\ \mathit{priv}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

To sign a message, Alice would use her private key:

$$\begin{aligned}&\mathit{sign} :: \mathit{PublicK} \to \mathit{PrivateK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{sign}\ \mathit{pub}\ \mathit{priv}\ m = (m \uparrow (\mathit{privD}\ \mathit{priv}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

Signing, hence, is just the same as decryption, but on a message that was not yet encrypted. To verify the signature, Bob uses Alice's public key with a function similar to encryption:

$$\begin{aligned}&\mathit{verify} :: \mathit{PublicK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{verify}\ \mathit{pub}\ s = (s \uparrow (\mathit{pubE}\ \mathit{pub}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

In this form, RSA is not safe, however. There are a lot of issues that must be addressed. In particular, we omitted everything related to *padding* messages before they are encrypted and to *hashing* messages before they are signed. These steps are essential for RSA to be secure. But since these steps have little relation to the mathematics of primes, they are not relevant to this chapter. After all, this is not an introduction to cryptography.

5.12. Open Problems

In the summer of 1742, Christian Goldbach (1690 – 1764), a mathematician and diplomat in the service of the Russian Czar, wrote a letter to Leonhard Euler. In this letter he told Euler of an observation he made: every even natural number greater than 2 can be represented as a sum of two primes. He had tested this with a lot of numbers – basically, with any number he could find – but was unable to prove the conjecture. Euler was excited about the observation and answered that he was sure that the conjecture must be true but that he was unable to demonstrate it either. This, basically, is still the state of affairs today: There is a lot of evidence that Goldbach's conjecture is true, much more than in the times of Euler and Goldbach, but there is no rigorous proof.

Let us see some examples:

$4 = 2 + 2,$
$6 = 3 + 3,$
$8 = 3 + 5,$
$10 = 3 + 7 = 5 + 5,$
$12 = 5 + 7,$
$14 = 7 + 7 = 3 + 11,$
$16 = 3 + 13$

and so on.

Here is a function to compute for any number n one of the sums of two primes that equal n:

```
goldbach :: Natural → (Natural, Natural)
goldbach n | odd n      = error "not even"
           | n ≡ 2      = error "not greater 2"
           | otherwise = go allprimes
  where go [] = error "The end is ny!"
        go (p : ps) | p > n − 2 = error ("disproved: " ++ show n)
                    | otherwise = let q = n − p
                                  in if prime q then (p, q)
                                     else go ps
```

The really interesting point is the second error in the *go* function. When you can find a number, for which this error occurs, there is certainly some mathematical honour you can earn.

One can think of many ways to find numbers to which to apply the *goldbach* function, the result is always a prime tuple, *e.g.* powers of 2:

$(2, 2), (3, 5), (3, 13), (3, 29), (3, 61), (19, 109), (5, 251), (3, 509), (3, 1021),$

the year of the first known formulation of the conjecture: *goldbach* $1742 = (19, 1723)$, safe primes: *map goldbach* (*map* (+1) *safeprimes*):

$(7,257),(11,337),(7,353),(5,379),(5,463),(13,467),(5,499),(7,557),(11,577),\dots$

or the ASCII code of the word “goldbach”:
goldbach \$ *read* \$ *concatMap* (*show* ∘ *ord*) `"goldbach"`: ... (This will take a while to compute, so let us continue ...)

There is so much evidence in favour of the correctness of the conjecture that it is considered to be true by most mathematicians today. But there is still no rigorous proof. We cannot claim the truth of the conjecture just by applying it to a finite set of numbers, even if that set is incredible large. The point is of course that there are infinitely many numbers. For any number n up to which we may have tested the property, there may be a larger number $n+k$ for which the property does not hold. By just computing concrete results for given numbers, we can therefore not prove the theorem; we can only disprove it this way by finding a number n for which the property does not hold.

The Goldbach property, *i.e.* the property of a number to be representable by the sum of two primes, is so prototypical for many problems in mathematics that properties of this kind are often called *Goldbach-like* in mathematical logic. Goldbach-like properties can be easily calculated for any given number n, but there is no way of proving that such a property holds for all numbers other than going through all of them and actually calculating it for each and every one. Since we cannot go through all numbers in a finite number of steps, this kind of statements, even though calculable for any given instance, cannot be calculated from the axioms of a given formal system.

There is nothing special about the Goldbach conjecture itself that would yield this characteristic. In fact, many unproven conjectures share it. That every number can be factored into primes, for instance, is a Goldbach-like statement too. For this statement, we actually have a simple proof. We even have a proof for the much stronger fundamental theorem of arithmetic that states that every number has a *unique* prime factorisation (which is not a Goldbach-like statement). If we did not have a proof of the fact that every number can be factored into primes, the only technique we would have at hand to prove or disprove the statement would be to find a counterexample. Testing numbers for this property as such would be quite simple, since we just have to factor given numbers and say whether they can be factored into primes or not. Until now, however, the property has turned out to be true for any number we have looked at. If we not had the proof of the theorem, we could imagine that there might be a counterexample lurking among the infintely many numbers we have not yet examined. But there are not enough computing resources to look at all of them in finite time.

A simple property that is not a Goldbach-like statement is the twin prime conjecture, which we already encountered, stating that there is an infinite number of pairs of primes p and q, such that $q = p + 2$. Examples are 3 and 5, 5 and 7, 11 and 13, 17 and 19, 29 and 31 and many other number pairs. Unlike the Goldbach conjecture, we cannot disprove the twin prime conjecture by finding a counterexample. The conjecture states that there are infinitely many twin primes. There is thus no criterion to conclude from the fact that a pair of numbers (n,m) does not have the property of being twin primes that the conjecture is false. We could, for instance, get excited about the fact that 23 has no twin and declare 17 and 19 the last pair of primes. However, when we go on, we will find 29 and 31. The only way to disprove the twin prime conjecture is therefore to go through all numbers, which, again, is not possible in a finite number of steps.

A similar is true for the fundamental theorem of arithmetic. We cannot decide this theorem even for a single number. For any number we factor, we additionally have to compare the obtained factorisation with those of all other numbers (including those we have not yet examined) to decide if the factorisation is unique. Establishing the property for a single number already includes infinitely many steps and is therefore not possible in practical terms. Apparently, there is no way to prove this theorem but in an indirect fashion.

Another complex of open problems is factoring itself for which, as we have seen, no efficient algorithm is known. There are ways to find the prime factors of a given number, of course. But with large numbers, factoring becomes resource-intensive. We need a lot of steps to find the factors.

The factoring problem is tightly coupled with the problem of solving the discrete logarithm. We got an idea of this coupling already, when we looked at the Carmichael theorem: when we know the factors of a number, we can easily find a number t, such that $a^t \equiv 1 \pmod{n}$, where a and n are coprime. Shor's quantum factoring algorithm, actually, exploits this coupling the other way round. It finds the number t and uses t to find two prime factors of n.

In the world of classic, non-quantum computing, we know of only one way to find t, the order of the group generated by a, namely to raise a to the sequence of numbers $1, 2, \dots, n-1$ until a raised to one of these numbers is $1 \pmod{n}$. Here is a Haskell function that uses this logic to find t for a given a:

```
order :: Natural → Natural → Natural
order n a = go 2 (a ∗ a)
  where go t a′ | a′ `rem` n ≡ 1 = t
                | t ⩾ n          = error "group exhausted"
                | otherwise      = go (t + 1) (a′ ∗ a)
```

Note that we introduce a guard that saves us from looping eternally: when we reach $t = n$, we abandon the function. We will see why we do this in a second.

Once we have found t, it is easy to find the factors. We start – once again – with the equation

$$a^t \equiv 1 \pmod{n} \tag{5.65}$$

and subtract 1 from both sides yielding

$$a^t - 1 \equiv 0 \pmod{n}. \tag{5.66}$$

Then we factor the left-hand side of the equation:

$$(a^{\frac{t}{2}} - 1)(a^{\frac{t}{2}} + 1) \equiv 0 \pmod{n}. \tag{5.67}$$

In other words, the product of $a^{\frac{t}{2}} - 1$ and $a^{\frac{t}{2}} + 1$ is congruent to 0 modulo n, *i.e.* this product is a multiple of n. That, in its turn, means that n and this product must have common factors. Consequently, $\gcd(a^{\frac{t}{2}} - 1, n)$ or $\gcd(a^{\frac{t}{2}} + 1, n)$ will produce at least one factor of n.

This does not work in all cases, however. First, t must be even; otherwise $\frac{t}{2}$ would not be natural number. If t is odd, we therefore have to look for another a. Second, a must be coprime to n. On the other hand, If a is not coprime to n, then we have already found a factor of n, namely the gcd of n and a. Third, $a^{\frac{t}{2}}$ should not be $n-1$, since in that case a^t is trivially $1 \pmod{n}$. Finally, n should be squarefree. If n is not squarefree, some numbers a will fulfil the equation $a^t \equiv 0 \pmod{n}$. That is, there are remainders of n that, multiplied by themselves, will yield a multiple of n. In that case, we will hit the error "group exhausted" in the *order* function above. The property of a number being squarefree or not, however, is hard to establish. If we introduce a test at the beginning of the algorithm, we are back where we started: an algorithm that takes an exponential number of steps. We therefore have to accept that an error may occur for any number on which we apply Shor's algorithm.

Here is a simple Haskell implementation of the classical part of Shor's algorithm using the order function defined above:

```
shorfact :: Natural → IO [Natural]
shorfact 0 = return [ ]
shorfact 1 = return [ ]
shorfact 2 = return [2]
shorfact n | even n     = do fs ← shorfact (n `div` 2)
                             return (2 : fs)
           | otherwise = do p ← rabinMiller 16 n
                             if p then return [n] else loop
  where loop = do a ← randomNatural (3, n − 2)
                  case gcd n a of
                    1 → check a (order n a)
                    f → do fs1 ← shorfact (n `div` f)
                           fs2 ← shorfact f
                           return (fs1 ++ fs2)
        check a t  | odd t = loop
                   | (a ↑ (t `div` 2)) `rem` n ≡ n − 1 = loop
                   | otherwise = let f1 = gcd n (a ↑ (t `div` 2) − 1)
                                     f2 = gcd n (a ↑ (t `div` 2) + 1)
                                     f | f1 ≡ 1     = f2
                                       | otherwise = f1
                                 in do fs2 ← shorfact (n `div` f)
                                       fs1 ← shorfact f
                                       return (fs1 ++ fs2)
```

As usual, we start with some trivial base cases: 0 and 1 have no factors; 2 is the first prime and has only one factor, namely 2 itself. Then, if n is even, we apply the algorithm to half of n and add 2 to the resulting list of factors. Otherwise, we first check if n is prime using the Rabin-Miller test (with a relaxed repetition value). If it is prime, there is only one factor, namely n. Otherwise, we enter *loop*.

Here, we start by choosing a random number that is not in one of the trivial groups of n and test if we have been lucky by checking if $\gcd(n, a) = 1$. If it is not 1, then we have found a factor by chance. We continue with *shorfact* on n dividied by f and on f, which still could be a composite factor of n. Finally, we merge the two resulting lists of factors.

Otherwise, we call *check*. This function decides how to continue depending on the result of *order*: we may need to start again, if t does not fulfil the preconditions, or we continue with the gcds. If t is odd, a turns out to be useless and we start again with another a. If $a^{\frac{t}{2}}$ is $n-1$, a is again useless and we start with another a. Otherwise, we compute the gcds of $a^{\frac{t}{2}} \pm 1$. We take one of the results to continue just making sure that, should one of the values be 1, we take the other one. That is a bit sloppy of course, since the case that both results are 1 is not handled. If that happens, however, something must be wrong in our math and, then, we cannot guarantee that the result is correct at all.

The quantum magic enters in the *order* subroutine. The algorithm that finds the order of a uses techniques that are far from what we have learnt so far, in particular *Fourier analysis.* Fourier analysis is a technique that represents complicated functions in terms of simpler and well-understood functions, namely trigonometric functions. The idea is that the quantum processor is initialised with a *superposition* of the period of the function $f(x) = x^t \bmod n$, which is then simplified reducing f by Fourier analysis.

This sounds like alchemy – and, yes, basically it is alchemy, quantum alchemy to be specific. We will look at Fourier analysis later, when we have studied the calculus. But I cannot promise that you will understand quantum alchemy when you will have understood Fourier. I am not sure, in fact, if even quantum alchemists understand quantum alchemy.

The result of the quantum Fourier analysis, as always in quantum computing, is correct with a certain probability. The whole algorithm, therefore, must be repeated and we must ensure that there is one result that appears significantly more often than others. The reasoning here is similar to the reasoning we have already applied to the Rabin-Miller test. So, this part of the algorithm (which we have left out above) should not be shocking. Shocking is rather the fact that there actually is a quantum algorithm that solves mathematical problems that cannot be solved (yet) on classic computers. This may be a hard boundary that cannot be passed, so that we have to accept that there are problems that can be solved in the classic world and others than can be solved only in the quantum world. This is an open and, indeed, very deep question in modern mathematics and computer science. But let us come back to primes, which, for my taste, are spooky enough.

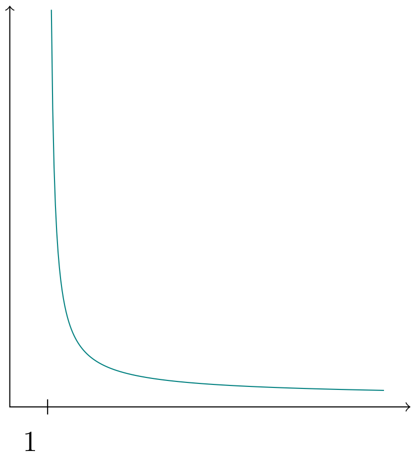
The most fundamental unsolved problems deal with the distribution of primes and the *prime number theorem.* To investigate the distribution of primes, we can start by counting primes up to a given number n, a function usually called $\Pi(n)$. We can implement Π in Haskell as:

```
countp :: Natural → Int
countp n = length (takeWhile (<n) allprimes)
```

$\Pi(10)$ or, in Haskell, *countp* 10 gives 4, since there are 4 primes less than 10: 2, 3, 5 and 7. Here are the values for some small powers of 10:

$$\begin{array}{lcr} 10^1 & = & 4 \\ 10^2 & = & 25 \\ 10^3 & = & 168 \\ 10^4 & = & 1229 \\ 10^5 & = & 9592 \end{array}$$

Such tables of values of Π were already compiled in the early 18th century. From some of such tables Legendre conjectured that Π is somehow related to the natural logarithm (a beast we already met without much explanations in the first chapter). Specifically, he proposed that $\Pi(n) = \frac{n}{\ln(n)+c}$, where c is some constant. Gauss and later his student Peter Gustav Lejeune Dirichlet (1805 – 1859) improved this conjecture with the *logarithmic integral*, a concept from calculus and, as such, far ahead on our way. Intuitively, the integral yields the area under a curve; the logarithmic integral is the area under the curve described by the function $\frac{1}{\ln(n)}$, which looks like this:



Let us compare the precision of these two approximations. The following table lists the values for $\Pi(n)$ and the differences $\frac{n}{\ln(n)} - \Pi(n)$ and $li(n) - \Pi(n)$, where $li(n)$ is the logarithmic integral of n:

n	$\Pi(n)$	$n/\ln(n)$	$li(n)$
10	4	0	2
10^2	25	-3	5
10^3	168	-23	10
10^4	1229	-143	17
10^5	9592	-906	38
10^6	78498	-6116	130
10^7	664579	-44158	339
10^8	5761455	-332774	754
10^9	50847534	-2592592	1701
...	...	...	...

The error of the logarithmic integral grows much slower than that of Legendre's conjecture. For small numbers, the error of the li variant is still greater. But already for $n = 10^3$, Legendre overtakes li and is then increasing orders of magnitude faster than li.

This is not the last word, however. There is a function that is still more precise in most cases. In his famous paper, "On the Number of Primes less than a given Magnitude", the ingenious mathematician Bernhard Riemann (1826 – 1866) not only introduced the most tantalizing of all unsolved mathematical problems, the *Riemann Hypothesis*, but he also proposed a refinement to Gauss/Dirichlet's Π-approximation. He conjectured that Π corresponds roughly to the infinite series

$$R(n) = li(n) - \frac{1}{2}li(n^{\frac{1}{2}}) - \frac{1}{3}li(n^{\frac{1}{3}}) - \frac{1}{5}li(n^{\frac{1}{5}}) + \frac{1}{6}li(n^{\frac{1}{6}}) - \dots \tag{5.68}$$

That the number of primes is related to the natural logarithm is already an astonishing fact. But, now, Riemann goes even further. To see what it is that is so surprising about Riemann's improvement, we present the equation above in a more compact form:

$$R(n) = \sum_{k=1}^{\infty} \frac{\mu(n)}{k} li(n^{\frac{1}{k}}) \tag{5.69}$$

The function $\mu(n)$ appearing in the formula is the Möbius function, which yields 0, 1 or -1 depending on n being squarefree and the number of prime factors of n being even or odd. To remind you of the first values of the Moebius function: $\mu(1) = 1$, $\mu(2) = -1$, $\mu(3) = -1$, $\mu(4) = 0$, $\mu(5) = -1$ and $\mu(6) = 1$ leading to the first terms of the summation above: $\frac{1}{1}li(n^{\frac{1}{1}}) = li(n)$, $\frac{-1}{2}li(n^{\frac{1}{2}})$, $\frac{-1}{3}li(n^{\frac{1}{3}})$, $\frac{0}{4}li(n^{\frac{1}{4}}) = 0$, $\frac{-1}{5}li(n^{\frac{1}{5}})$ and $\frac{1}{6}li(n^{\frac{1}{6}})$. In other words: the Möbius function is intimately related to the concept of the distribution of primes through the prime number theory, even if it does not reveal any regularity at the first and even the second sight.

For many values of n, the Riemann refinement is much closer to the real value of Π than either Legendre's try and Gauss/Dirichlet's improvement. It is esitmated that it is better 99% of the time. Occasionally, however, Riemann's value is worse than that of Gauss/Dirichlet. The latter fact is known only theoretically, no specific n is known that makes Riemann worse. In most cases, Riemann's value is even much better, for instance: for 10^9, Gauss/Dirichlet's deviation is 1701 while Riemann's difference is only -79; The difference of Gauss/Dirichlet for 10^{16} is more than 3 million; Riemann's difference is 327 052 and thus 10 times better.

The prime number theorem is usually stated as an asymptotic law of the form:

$$\lim_{n\to\infty} \frac{\Pi(n)}{n/\ln(n)} = 1, \tag{5.70}$$

which means that for large n, the relation of $\Pi(n)$ and $\frac{n}{\ln(n)}$ approximates 1. A first attempt to prove the theorem was made by the Russian mathematician Pafnuty Lvovich Chebyshev (1821 – 1894). Even though his paper failed to strictly prove the theorem, he could prove Bertrand's postulate, which states that there is at least one prime number between n and $2n$ for $n \geq 2$ and on which we have already relied when looking at cryptography. A proof was finally given at the the end of the 19^{th} century independently by the French mathematicians Jacques Hadamard (1865 – 1963) and Charles Jean de la Vallée-Poussin (1866 – 1962).

It is unknown until today, though, if strict error margins for the approximation of Π can be given and if the available approximations can be further improved. Many of these questions are related to Riemann's Hypothesis. But mathematicians today appear to be far from producing a proof (or refutation) of this mega-hypothesis.

6. The Inverse Element

6.1. Inverse Elements

In a group, any element a has an inverse a', such that $a \cdot a' = e$, where e is the neutral element of that group. We have already met some groups, permutation groups and finite groups of modular arithmetic. We will now look at infinite groups, namely the additive group over $\mathbb{Z}$, the integers, and the multiplicative group over $\mathbb{Q}$, the rationals.

As you may have guessed already, the inverse element of a natural number n in the additive group over integers is the *negation* of n, $-n$. It is easy to see that n plus its inverse, $-n$, is just 0, the neutral element of addition: $n+(-n) = n-n = 0$. Since it is, of course, also true that $-n+n = n-n = 0$, the inverse element of a negative number $-n$ is its positive counterpart n. With the law of double negation, $-(-n) = n$, that is the negation of the negation of n is n, the formula $n + (-n) = 0$ is universally true for both positive and negative numbers. When we are talking about n, the positive, we just have : $n + (-n) = n - n = 0$. When we are talking about $-n$, the negative, we have: $-n + (-(-n)) = -n + n = n - n = 0$.

Note that with this identity, we can solve any equation of the form $a + x = b$ in the additive group of $\mathbb{Z}$. We just add the inverse of a to both sides of the equation: $a + (-a) + x = b + (-a)$, which, of course, is $x = b - 1$. Without negative numbers, there were equations we could not solve in this way, for instance $4 + x = 3$. We had gaps, so to speak, in our additive equations. But now, in the group over integers, there are no such gaps anymore: we just add the inverse of 4 on both sides and get $x = 3 - 4 = -1$.

What is the inverse element of integers in the multiplicative group? Well, in this group, it must still hold that $a \cdot a' = e$, where $\cdot$ is multiplication and e, the neutral element, is unity. We, hence, have $a \times a' = 1$. We easily find a solution with division. We divide a on both sides and get $a' = \frac{1}{a}$ and that is the answer: the inverse element of a natural number n in the multiplicative group is $\frac{1}{n}$.

We see that we now can solve any multiplicative equation of the form $ax = b$, just by multiplying the inverse of a on both sides: $ax\frac{1}{a} = b\frac{1}{a}$, which is $x\frac{a}{a} = \frac{b}{a}$. The left-hand side reduces to $1x = x$ and we have $x = \frac{b}{a}$. As we have already seen in finite groups, prime numbers simply "disappear" with fractions, since we can now reach them with multiplication, for instance: $3x = 5$ is equivalent to $x = \frac{5}{3}$.

Let us go on and build something bigger: a field consisting of addition and multiplication where the distributive law holds. We already know that the distributive law holds in the world of natural numbers: $a(b + c) = ab + ac$. But what, when we have creatures like $-n$ and $\frac{1}{n}$? We obviously need rules to add fractions and to negate products.

Let us start with fractions. We may add fractions with the same denominator by simply adding the numerators, *i.e.* $\frac{a}{n} + \frac{b}{n} = \frac{a+b}{n}$. With different denominators, we first have to manipulate the fractions in a way that they have the same denominator, but without changing their value. The simplest way to do that is to multiply one denominator by the other and to multiply the numerators correspondingly: $\frac{a}{n} + \frac{b}{m} = \frac{am}{nm} + \frac{bn}{mn} = \frac{am+bn}{mn}$.

We can reduce the computational complexity of this operation in most cases by using the *lcm* instead of the product nm. As you may remember, the *lcm* of two numbers, a and b is $\frac{ab}{\gcd(a,b)}$. We would still multiply the numerator by the value by which the denominator changes, *i.e.* the *lcm* divided by the denominator and would get

$$\frac{a}{n} + \frac{b}{m} = \frac{a \times \frac{lcm(n,m)}{n} + b \times \frac{lcm(n,m)}{m}}{lcm(n,m)}. \tag{6.1}$$

That looks complicated, but is simple when we take two concrete numbers: $\frac{1}{6} + \frac{2}{9}$. We first compute the $lcm(6, 9)$ as

$$\frac{6 \times 9}{\gcd(6, 9)}.$$

The gcd of 6 and 9 is 3, the product $6 \times 9 = 54$ and $54/3 = 18$. So, we have:

$$\frac{1 \times \frac{18}{6} + 2 \times \frac{18}{9}}{lcm(6,9) = 18} = \frac{3 + 4}{18} = \frac{7}{18}.$$

It looks simpler now, but still it seems that we do need more steps than by just multiplying the respective other denominator to numerator and denominator of both fractions. However, when we do this, we have to operate with greater numbers and, at the end, reduce the fractions to their *canonical form*, which is

$$\frac{a}{b} = \frac{a/\gcd(a,b)}{b/\gcd(a,b)}. \tag{6.2}$$

For the example, this would mean

$$\frac{1 \times 9}{6 \times 9} + \frac{2 \times 6}{9 \times 6} = \frac{9}{54} + \frac{12}{54} = \frac{21}{54}.$$

Now, we reduce to canonical form:

$$\frac{21/\gcd(21,54)}{54/\gcd(21,54)},$$

which is

$$\frac{\frac{21}{3}=7}{\frac{54}{3}=18}.$$

How to multiply negative numbers of the form $-a(b+c)$? Let us look at an example: the additive inverse of 6 is -6. We would therefore expect $-6+6=0$. Furthermore, we have $2\times 3=6$. Now, if we add the inverse of 3 to 3 once, we get 0: $-3+3=0$. What should we get, if we add the inverse of 3 twice to twice 3, *i.e.* $2\times 3+2\times -3$? We expect it to be 0, correct? Therefore and since 2×3 is 6, 2×-3 must be the inverse of 6, hence, -6.

The same is true, the other way round, thus $2\times 3+(-2)\times 3=0$. That means that we can move the minus sign in a product, such that $a\times -b=-a\times b$. To any such product we can simply add the factor 1 without changing the result: $a\times b=1\times a\times b$. We, therefore, have $1\times a\times -b=1\times -a\times b=-1\times a\times b$. This facilitates life a bit: we can handle one minus sign as the additional factor -1. In other words, multiplying by -1 has the same effect has negating: $-1n=-n$.

Going back to the question of how to handle products of the form $-a(b+c)$, we now can say that $-a(b+c)=-1a(b+c)$. Multiplying a out in terms of the distributive law, we get: $-1(ab+ac)$. Now, we multiply -1, just as we did above: $(-1)ab+(-1)ac$ and, since we know that $-1n=-n$, we derive $-ab-ac$.

What if we have more than one minus sign in a product like, for instance: -2×-3? We saw above that $2\times -3+2\times 3=0$ and we can reformulate this as $-1\times 2\times 3+2\times 3=0$. Now, if we have two negative factors, we add one more minus sign, *i.e.* one more factor -1: $-1\times -1\times 2\times 3+2\times 3=?$ We just substitute one -1 after the other by negation. We first get $-1\times -(2\times 3)\dots$ and then $-(-(2\times 3))$. We now see clearly that this should be the negation, *i.e.* the inverse of $-(2\times 3)$. The inverse of $-(2\times 3)$, however, is just 2×3 and that is 6. Therefore: $-2\times -3+2\times 3=12$.

So, what happens if we multiply negative numbers and fractions? When we just follow multiplication rules we get $-1\times\frac{1}{n}=\frac{-1}{n}$. We, hence, would say, according to the rules derived above, that $\frac{-1}{n}$ is the additive inverse of $\frac{1}{n}$. What about $\frac{1}{-n}$? This should be the multiplicative inverse of $-n$, such that $-n\times\frac{1}{-n}=1$. We again can move the minus sign around to get $-1\times n\times\frac{1}{-n}$ leading to $-1\times\frac{n}{-n}=1$. Dividing n on both sides gives $-1\times\frac{1}{-n}=\frac{1}{n}$ and multiplying -1 gives $\frac{1}{-n}=\frac{-1}{n}$. In other words, a fraction with a minus sign in it, independently of where it appears, in the numerator or the denominator, is the additive inverse of the same fraction without the minus sign: $\frac{a}{b}+\frac{-a}{b}=\frac{a}{b}+\frac{a}{-b}=\frac{a}{b}-\frac{a}{b}=0$.

In summary, we can define a set of rules on multiplication that are independent of whatever a and b are:

$$\begin{array}{rcrcr}
a & \times & b & = & ab \\
-a & \times & -b & = & ab \\
-a & \times & b & = & -ab \\
a & \times & -b & = & -ab
\end{array}$$

With this, we have established an important theoretical result, namely that $\mathbb{Q}$, including negative numbers, is an infinite field with addition and multiplication. But let us go on. There are still operations we have seen for natural numbers combining multiplication and addition that may and should have an interpretation with integers and fractions, namely exponentiation.

From the table above, we see immediately that products with an even number of negative numbers are positive – a fact that we already used when discussing prime numbers. In general, any number raised to an even exponent is positive independent of that number being positive or negative.

This leads to a difficulty with the root operation, since even roots may have two different results: a positive number or its additive inverse. For instance, $\sqrt{4}$ could be 2 and -2. Even further, the operation cannot be applied to a negative number: $\sqrt{-1}$ has no meaning – at least not with the creatures we have met so far. There may be an object i that fulfil equations of the form $i = \sqrt{-1}$, but such objects are beyond our imagination at this point in time.

Now, what is the effect of having negative numbers or fractions in the exponent?

We will first look at fractions as exponents and investigate powers of the form $a^{\frac{1}{n}}$. To clarify the meaning of such expression, we will use the rules we know so far to observe what happens, when we multiply two powers with the same base and with fractional exponents:

$$x = a^{\frac{1}{n}} \times a^{\frac{1}{m}} = a^{\frac{1}{n}+\frac{1}{m}}. \tag{6.3}$$

Let us look at the special case of the exponent $\frac{1}{2}$:

$$x = a^{\frac{1}{2}} \times a^{\frac{1}{2}} = a^{\frac{1}{2}+\frac{1}{2}}. \tag{6.4}$$

Obviously, $\frac{1}{2} + \frac{1}{2} = 2 \times \frac{1}{2} = 1$. In other words, x, in this case, is just a. Furthermore, we see that there is a number $n = a^{\frac{1}{2}}$, such that $n \times n = n^2 = x$. For which number does this hold? Well, it is just the definition of the square root: $\sqrt{x} \times \sqrt{x} = (\sqrt{x})^2 = x$. We would conclude that $a^{\frac{1}{n}}$ is equivalent to $\sqrt[n]{a}$:

$$a^{\frac{1}{n}} = \sqrt[n]{a}. \tag{6.5}$$

This, in fact, makes a lot of sense. We would expect, for instance, that $(a^{\frac{1}{n}})^n$ is just a, since $(\sqrt[n]{a})^n = a$. When we multiply it out, we get indeed $a^{\frac{1}{n} \times n} = a^1 = a$. We would also expect that $(a^{\frac{1}{n}})^{\frac{1}{m}}$ is $\sqrt[m]{\sqrt[n]{a}} = \sqrt[mn]{a}$, *e.g.* $(a^{\frac{1}{2}})^{\frac{1}{2}} = \sqrt{\sqrt{a}} = \sqrt[4]{a}$. When we multiply it out again, we indeed obtain $a^{\frac{1}{2} \times \frac{1}{2}} = a^{\frac{1}{4}}$.

Now, what about negative exponents? We adopt the same technique, *i.e.* we multiply two powers with the same base:

$$a \times a^{-1}.$$

We can write this as $a^1 \times a^{-1}$ and this is

$$a^1 \times a^{-1} = a^{1-1} = a^0 = 1.$$

We see a^{-1} is the multiplicative inverse of a. But we already know that the inverse of a is $\frac{1}{a}$. We conclude that

$$a^{-n} = \frac{1}{a^n}. \tag{6.6}$$

This conjecture would imply that $a^{-n} \times a^n = 1$, since $\frac{1}{a^n} \times a^n = 1$ and, indeed: $a^{-n} \times a^n = a^{-n+n} = a^0 = 1$. It would also imply that $(a^{-n})^{-1} = a^n$, since $a^{-n} = \frac{1}{a^n}$, whose inverse is a^n. Indeed, we have $(a^{-n})^{-1} = a^{-n \times -1} = a^n$.

A side effect of this rule is that we now have a very nice notation for the multiplicative inverse. Until now, we have used the symbol a' to denote the inverse of a. Since $'$ is also used in other contexts, the notation a^{-1} is much clearer and we will stick to it from now on.

6.2. $\mathbb{Z}$

We will now implement a number type that takes signedness into account. We will do so in a way that allows us to negate objects of different kind, basically any type of number. We therefore start by defining a parametrised data type:

```
data Signed a = Pos a | Neg a
  deriving (Eq, Show)
```

The data type has two constructors, *Pos* and *Neg* for a positive and a negative *a* respectivley. The expression **let** x = *Neg* 1 would assign the value -1 to x. We would instantiate a concrete data type by giving a concrete type for the type parameter, *e.g.*

```
type Zahl = Signed Natural
```

This type is called *Zahl*, the German word for *number*, which was used for the designation of the set $\mathbb{Z}$ of the integers. When Abstract Algebra started to be a major field of mathematics, the University of Göttingen was the gravitational centre of the math world and, since it was not yet common to use English in scientific contributions, many German words slipped into math terminology.

For convenience, we add a direct conversion from *Zahl* to *Natural*:

```
z2n :: Zahl → Natural
z2n (Pos n) = n
z2n (Neg _) = ⊥
```

Another convenience function should be defined for *Neg*, namely to guarantee that 0 is always positive. Otherwise, we could run into situations where we compare *Pos* 0 and *Neg* 0 and obtain a difference that does not exist. We therefore define

```
neg0 :: (Eq a, Num a) ⇒ a → Signed a
neg0 0 = Pos 0
neg0 x = Neg x
```

We now make *Signed* an instance of *Ord*:

```
instance (Ord a) ⇒ Ord (Signed a) where
  compare (Pos a) (Pos b) = compare a b
  compare (Neg a) (Neg b) = compare b a
  compare (Pos _) (Neg _) = GT
  compare (Neg _) (Pos _) = LT
```

The difficult cases are implemented in the first two lines. When a and b have the same sign, we need to compare a and b themselves to decide which number is greater than the other. If the sign differs, we can immediately decide that the one with the negative sign is smaller.

Signed is also an instance of *Enum*:

```
instance (Enum a) ⇒ Enum (Signed a) where
  toEnum i | i ⩾ 0    = Pos $ toEnum i
           | i < 0    = Neg $ toEnum i
  fromEnum (Pos a) = fromEnum a
  fromEnum (Neg a) = negate (fromEnum a)
```

With this definition some more semantics comes in. We explicitly define that, converting an integer greater or equal to zero, we use the *Pos* constructor; converting an integer less than zero, we use the *Neg* constructor. Furthermore, when we convert in the opposite direction, *Pos* a is just an a, whereas *Neg* a is the negation of a.

Now we come ot arithmetic, first addition:

```
instance (Ord a, Num a) ⇒ Num (Signed a) where
  (Pos a) + (Pos b) = Pos (a + b)
  (Neg a) + (Neg b) = neg0 (a + b)
  (Pos a) + (Neg b) | a > b = Pos (a - b)
                    | a < b = Neg (b - a)
                    | a ≡ b = Pos 0
  (Neg a) + (Pos b) | a > b = Neg (a - b)
                    | a < b = Pos (b - a)
                    | a ≡ b = Pos 0
```

The addition of two positive numbers is a positive sum. The addition of two negative numbers ($-a + (-b) = -a - b$) is a negative sum. The addition of a negative and a positive number results in a difference, which may be positive or negative depending on which number is greater: the negative or the positive one.

We define subtraction in terms of addition: subtracting a positive number b from any number a is the same as adding the negation of b to a. Vice versa, subtracting a negative number b is the same as adding a postive number.

```
a - (Pos b) = a + (neg0 b)
a - (Neg b) = a + (Pos b)
```

Multiplication:

```
(Pos a) * (Pos b) = Pos (a * b)
(Neg a) * (Neg b) = Pos (a * b)
(Pos 0) * (Neg _) = Pos 0
(Pos a) * (Neg b) = Neg (a * b)
(Neg _) * (Pos 0) = Pos 0
(Neg a) * (Pos b) = Neg (a * b)
```

This is a straight forward implementation of the rules we have already seen above: the product of two positive numbers is positive; the product of two negative numbers is positive; the product of a positive and a negative number is negative.

The next method is *negate*. There is one minor issue we have to handle: what do we do if the number is 0? In this case, we assume the number is positive. But that is a mere convention. Without this convention, we would have to introduce a constructor for 0 that is neither postive nor negative.

```
negate (Pos a) | signum a ≡ 0 = Pos a
               | otherwise    = Neg a
negate (Neg a) = Pos a
```

Finally, we have *abs*, *signum* and *fromInteger*. There is nothing new in the implementation of these methods:

```
abs (Pos a)      = Pos a
abs (Neg a)      = Pos a
signum (Pos a) = Pos (signum a)
signum (Neg a) = Neg (signum $ negate a)
fromInteger i   | i ⩾ 0 = Pos (fromInteger i)
                | i < 0 = Neg (fromInteger $ abs i)
```

We make *Signed* an instance of *Real*:

```
instance (Real a) ⇒ Real (Signed a) where
  toRational (Pos i) = toRational i
  toRational (Neg i) = negate (toRational i)
```

We also make *Signed* an instance of *Integral*:

```
instance (Enum a, Integral a) ⇒ Integral (Signed a) where
  quotRem (Pos a) (Pos b) = let (q, r) = quotRem a b in (Pos q, Pos  r)
  quotRem (Neg a) (Neg b) = let (q, r) = quotRem a b in (Pos q, neg0 r)
  quotRem (Pos a) (Neg b) = let (q, r) = quotRem a b in (neg0 q, Pos r)
  quotRem (Neg a) (Pos b) = let (q, r) = quotRem a b in (neg0 q, neg0 r)
  toInteger (Pos a) = toInteger a
  toInteger (Neg a) = negate (toInteger a)
```

The implementation of *toInteger* contains nothing new. But have a look at the definition of *quotRem*. The first case, where both numbers are positive, is easy: we return a positive quotient and a positive remainder. When both numbers are negative, the quotient is postive and the remainder is negative. Indeed, when we have the equation $a = qb + r$ and both, a and b, are negative, then a postive q will bring b close to a. With $a = -5$ and $b = -2$, for instance, the quotient would be 2: $2 \times -2 = -4$. Now, what do we have to add to -4 to reach -5? Obviously, -1. Therefore the remainder must be negative.

Now, when we have a postive a and a negative b, for instance: $a = 5$ and $b = -2$; then a negative quotient will bring us close to a: $-2 \times -2 = 4$. Missing now is the positive remainder 1. Finally, when a is negative and b is positive, we need a negative quotient, *e.g.*: $a = -5$ and $b = 2$; then $-2 \times 2 = -4$. Missing, in this case, is again a negative remainder, namely -1.

In the future, there may arise the need to make number types signed that do no fit into the classes from which we derived *Signed* so far. In particular, a signed fraction should inherit from Fractional. We therefore make *Signed* an instance of Fractional:

```
instance (Eq a, Ord a, Fractional a) ⇒ Fractional (Signed a) where
  (Pos a) / (Pos b) = Pos (a / b)
  (Neg a) / (Neg b) = Pos (a / b)
  (Pos 0) / (Neg b) = Pos (0 / b)
  (Pos a) / (Neg b) = Neg (a / b)
  (Neg a) / (Pos b) = Neg (a / b)
```

6.3. Negative Binomial Coefficients

The aim of this section is to generalise binomial coefficients to integers using the new data type *Zahl*. There are many ways how to approach this goal, which perhaps can be grouped into two main approaches: first, we can look for an application of such a generalisation in the "real world" and search an appropriate mathematical tool for this problem. We started with binomial coefficients by discussing combinatorial issues, but, of course, combinatorial problems like the ways to choose k objects out of n provide no meaningful interpretation for negative numbers – what should a negative number of possibilities mean? But there are other applications, such as the multiplication of sums.

The second basic approach is not to look for applications, but to investigate the formalism asking "what happens, when we change it?" This may appear much less interesting at the first sight, since there is no obvious use case for such an altered formalism. However, such formal approaches do not only help deepening the insight into specific mathematical problems, but they also lead to new tools, which may find their applications in the future. This has happened more than once in the history of mathematics. When David Hilbert, the champion of the formalistic approach, redefined geometry extending it from the two-dimensional plane and the three-dimensional space to n-dimensional manifolds, there was no concrete application in sight. It took only a short while, however, before John von Neumann and others started using the concepts of Hilbert's geometry to model quantum physics.

Anyhow, we start with the second approach. Still, there are many ways to go. We can start with the formula $\binom{n}{k}$ and ask ourselves what happens if $n < 0$ or $k < 0$. To begin with, let us assume $n < 0$ and $k \geq 0$. If we just apply the formula $\frac{n(n-1)\dots(n-k+1)}{k!}$, we get for $\binom{-n}{k}$ a kind of falling factorial in the numerator that looks like a rising factorial with minus signs in front of the numbers, *e.g.* $\binom{-6}{3}$ is $\frac{-6\times-7\times-8}{6}$, which is $-1\times-7\times-8 = -56$. Obviously, the signedness of the result depends on whether k is even or odd. Since there are k negative factors in the numerator, the product will be positive if k is even and negative otherwise.

Here is a Haskell implementation for this version of negative binomial coefficients:

```
chooseNeg :: Zahl → Natural → Zahl
chooseNeg n k | n ⩾ 0 ∧ k ⩾ 0 = Pos (choose (z2n n) k)
              | n ≡ k          = 1
              | k ≡ 0          = 1
              | k ≡ 1          = n
              | otherwise      = (n > | (Pos k)) ‘div‘ (fac (Pos k))
```

This function accepts two arguments, one of type *Zahl* and the other of type *Natural*. For the moment, we want to avoid negative *k*s and to rule negative values out right from the beginning, we choose *Natural* as data type for *k*.

When both, *n* and *k*, are positive, we just use the old *choose* converting *n* to *Natural*. Then we handle the trivial cases. Finally, we just implement one of the formulas for binomial coefficients. Here are some values:

```
map (chooseNeg (−1)) [0 . . 9]
[1, −1, 1, −1, 1, −1, 1, −1, 1, −1]
```

```
map (chooseNeg (−2)) [0 . . 9]
[1, −2, 3, −4, 5, −6, 7, −8, 9, −10]
```

```
map (chooseNeg (−3)) [0 . . 9]
[1, −3, 6, −10, 15, −21, 28, −36, 45, −55]
```

```
map (chooseNeg (−4)) [0 . . 9]
[1, −4, 10, −20, 35, −56, 84, −120, 165, −220]
```

```
map (chooseNeg (−5)) [0 . . 9]
[1, −5, 15, −35, 70, −126, 210, −330, 495, −715]
```

```
map (chooseNeg (−6)) [0 . . 9]
[1, −6, 21, −56, 126, −252, 462, −792, 1287, −2002]
```

We see that the coefficients for each n alternate between positive and negative values depending on k being even or odd. We also see that there is no limit anymore from which on the coefficients are all zero. In the original definition, we had $\binom{n}{k} = 0$ for $k > n$. But now we have to give up that rule, because for $n < 0$ and $k \geq 0$, $k > n$ trivially holds for all cases. In consequence, we lose the nice symmetry we had in the original triangle. Of course, we can restore many of the old characteristics by changing the definition of *chooseNeg* for negative ns to `(n| >(Pos k))‘div‘(fac (Pos k))`. In this variant, let us call it *chooseNeg2*, we use the rising factorial, such that the absolute values of the numbers in the numerator equal the numbers in the numerator for $n \geq 0$. For instance:

```
map (chooseNeg2 (−2)) [0 . . 9]
[1, −2, 1, 0, 0, 0, 0, 0, 0, 0]
```

```
map (chooseNeg2 (−3)) [0 .. 9]
[1, −3, 3, −1, 0, 0, 0, 0, 0, 0]
```

```
map (chooseNeg2 (−4)) [0 .. 9]
[1, −4, 6, −4, 1, 0, 0, 0, 0, 0]
```

```
map (chooseNeg2 (−5)) [0 .. 9]
[1, −5, 10, −10, 5, −1, 0, 0, 0, 0]
```

```
map (chooseNeg2 (−6)) [0 .. 9]
[1, −6, 15, −20, 15, −6, 1, 0, 0, 0]
```

The first solution, *chooseNeg*, however, is more faithful to the original definition of the binomial coefficients, even if its results do not resemble the original results. One of the characteristics that is preserved is Pascal's rule:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \tag{6.7}$$

Indeed, we could use Pascal's rule to find negative coefficients in the first place. If we are interested in the coefficient $\binom{n-1}{k}$, we just subtract $\binom{n-1}{k-1}$ from both sides and get

$$\binom{n-1}{k} = \binom{n}{k} - \binom{n-1}{k-1}. \tag{6.8}$$

We can use this equation to search the unknown territory of negative coefficients starting from the well-known territory of positive coefficients using each result as a base camp for further expeditions. We start with $\binom{0}{0}$ and want to know the value for $\binom{-1}{0}$. Since coefficients for $k = 0$ are defined as 1, this case turns out to be trivial: $\binom{-1}{0} = 1$. The next is $\binom{-1}{1}$, which is $\binom{0}{1} - \binom{-1}{0}$, hence $0 - 1 = -1$. The next is $\binom{-1}{2} = \binom{0}{2} - \binom{-1}{1}$, which is $0 - (-1) = 0 + 1 = 1$. Next: $\binom{-1}{3} = \binom{0}{3} - \binom{-1}{2}$, which is $0 - 1 = -1$ and so on. Indeed, the coefficients of -1, as we have seen before, are just alternating positive and negative 1s:

```
map (chooseNeg (−1)) [0 .. 9]
[1, −1, 1, −1, 1, −1, 1, −1, 1, −1]
```

On the basis of this result, we can investigate the coefficients of -2. We know that $\binom{-2}{0}$ is 1 and continue with $\binom{-2}{1}$, which is $\binom{-1}{1} - \binom{-2}{0}$, which is $-1 - 1 = -2$. The next is $\binom{-2}{2} = \binom{-1}{2} - \binom{-2}{1}$ or $1 - (-2) = 1 + 2 = 3$. We continue with $\binom{-2}{3} = \binom{-1}{3} - \binom{-2}{2}$, which is $-1 - 3 = -4$. It turns out that the coefficients for $n = -2$ are

```
map (chooseNeg (−2)) [0 .. 9]
[1, −2, 3, −4, 5, −6, 7, −8, 9, −10],
```

which is a beautiful result. If we go on this way, we will reproduce the values observed above using *chooseNeg*.

Now, what about negative ks? There is no direct way to implement something like *chooseNeg* for negative ks, because we do not know what a negative factorial would mean. Of course, we can apply a trick very similar to that we used for *chooseNeg2*, *i.e.* we compute the factorial not as $fac\ n = n * fac\ (n-1)$, but as $fac\ n = n * fac\ (n+1)$ going up towards zero. In this case, we also have to take care of the rising or falling factorial of the numerator. For instance, we can use the falling factorial with the inverse of k, such that the value of the numerator remains the same independent of k being positive or negative. The result would resemble that of *chooseNeg2*, *i.e.* we would have alternating positive and negative coefficients for $n > 0$ and positive coefficients for $n < 0$.

More interesting is using Pascal's rule to create coefficients for negative ks. But we have to be careful. In equation 6.8, we used a coefficient for $k-1$ to find the coefficient for k. This will not work. When we look for $\binom{n}{-k}$, we do not yet know the value for $\binom{n}{-(k+1)}$, since we are entering the territory of negative numbers from above. Therefore, we need a variant of equation 6.8. Instead of subtracting $\binom{n-1}{k-1}$ from Pascal's rule, we subtract the other term $\binom{n-1}{k}$ and get

$$\binom{n-1}{k-1} = \binom{n}{k} - \binom{n-1}{k}. \tag{6.9}$$

It is obvious that any coefficient resulting from a positive n and a negative k in this way equals 0, since any coefficient with $k = 0$ is 1. So, $\binom{n}{-1} = 1 - 1 = 0$. However, we also have $\binom{0}{k} = 0$ for any k and $\binom{n}{n} = 1$. For $\binom{-1}{-2}$, we therefore have $\binom{0}{-1} - \binom{-1}{-1} = 0 - 1 = -1$. For $\binom{-1}{-3}$, we get $0 - \binom{-1}{-2} = 0 - (-1) = 0 + 1 = 1$. The next coefficient $\binom{-1}{-4}$, foreseeably, is $0 - \binom{-1}{-3}$, which is $0 - 1 = -1$. Again, for $n = -1$, we get a sequence of alternating negative and positive ones.

For $\binom{-2}{k}$, we get $\binom{-2}{-2} = 1$, $\binom{-2}{-3}$ is $\binom{-1}{-2} - \binom{-2}{-2} = -1 - 1 = -2$. $\binom{-2}{-4}$ then is $\binom{-1}{-3} - \binom{-2}{-3} = 1 - (-2) = 3$. The next coefficient is $\binom{-2}{-5} = \binom{-1}{-4} - \binom{-2}{-3}$, which is $-1 - 3 = -4$ and so on. The binomial coefficients for -2 with negative ks, thus, is just the sequence $1, -2, 3, -4, 5, -6, 7, -8, 9, -10, \dots$, which we have already seen for positive ks above. The symmetry of the triangle, hence, is reinstalled. For coefficients with $n = -2$ and $k = -9, -8, \dots, -1, 0, 1, \dots, 7$, we get the sequence

$-8, 7, -6, 5, -4, 3, -2, 1, 0, 1, -2, 3, -4, 5, -6, 7, -8.$

To confirm this result, we implement the backward rule as

```
pascalBack :: Zahl → Zahl → Zahl
pascalBack n 0 = 1
pascalBack n k | k ≡ n      = 1
               | k ≡ 0      = 1
               | n ≡ 0      = 0
               | n > 0 ∧ k < 0 = 0
               | n > 0 ∧ k ⩾ 0 = Pos (choose (z2n n) (z2n k)
               | k < 0      = pascalBack (n + 1) (k + 1) − pascalBack n (k + 1)
               | otherwise  = pascalBack (n + 1) k       − pascalBack n (k − 1)
```

We first handle the trivial cases $n = k$, $k = 0$ and $n = 0$. When $n > 0$ and $k < 0$, the coefficient is 0. For n and k both greater 0, we use *choose*. For $k < 0$, we use the rule in equation 6.9 and for $n < 0$, with $k > 0$, we use the rule in equation 6.8. Now we map *pascalBack* for specific ns to a range of ks:

```
map (pascalBack (−1)) [−9, −8 . . 9]
[1, −1, 1, −1, 1, −1, 1, −1, 1, 1, −1, 1, −1, 1, −1, 1, −1, 1, −1]

map (pascalBack (−2)) [−9, −8 . . 9]
[−8, 7, −6, 5, −4, 3, −2, 1, 0, 1, −2, 3, −4, 5, −6, 7, −8, 9, −10]

map (pascalBack (−3)) [−9, −8 . . 9]
[28, −21, 15, −10, 6, −3, 1, 0, 0, 1, −3, 6, −10, 15, −21, 28, −36, 45, −55]

map (pascalBack (−4)) [−9, −8 . . 9]
[−56, 35, −20, 10, −4, 1, 0, 0, 0, 1, −4, 10, −20, 35, −56, 84, −120, 165, −220]

map (pascalBack (−5)) [−9, −8 . . 9]
[70, −35, 15, −5, 1, 0, 0, 0, 0, 1, −5, 15, −35, 70, −126, 210, −330, 495, −715]
```

The symmetry of positive and negative ks is not perfect. The coefficients for $n < 0$ and $n < k < 0$ are all 0. The sequence, in consequence, is mirrowed with a delay of $|n| - 1$ ks, such that the coefficient that corresponds to the coefficient $\binom{n}{k}$ is not $\binom{n}{-k}$, but rather $\binom{n}{n-k}$. For instance: $\binom{-2}{4} = 5 = \binom{-2}{-6}$, $\binom{-3}{5} = -21 = \binom{-3}{-8}$ and $\binom{-5}{2} = 15 = \binom{-5}{-7}$.

Looking at the numbers of negative ns, I have the strange feeling that I already saw those sequences somewhere. But where? These are definitely not the rows of Pascal's triangle, but perhaps something else? Let us look at the triangle once again:

$$\begin{array}{lccccccccccccccccccc}
0 &&&&&&&&&& 1 \\
1 &&&&&&&&& 1 && 1 \\
2 &&&&&&&& 1 && 2 && 1 \\
3 &&&&&&& 1 && 3 && 3 && 1 \\
4 &&&&&& 1 && 4 && 6 && 4 && 1 \\
5 &&&&& 1 && 5 && 10 && 10 && 5 && 1 \\
6 &&&& 1 && 6 && 15 && 20 && 15 && 6 && 1 \\
7 &&& 1 && 7 && 21 && 35 && 35 && 21 && 7 && 1 \\
8 && 1 && 8 && 28 && 56 && 70 && 56 && 28 && 8 && 1 \\
9 & 1 && 9 && 36 && 84 && 126 && 126 && 84 && 36 && 9 && 1
\end{array}$$

Now follow the diagonals from the upper vertex left and right downwards. On both sides you see first the 1s, then the counting numbers $1, 2, 3, \ldots$, then the sequence we saw with $\binom{-3}{k}$, then the sequence we saw with $\binom{-4}{k}$ and so on. In other words, if we turn the triangle by 90° counterclockwise, we obtain the sequences for negative n; the same also works with a clockwise turn, but then we read the sequence from right to left.

This boils down to the equations

$$\left|\binom{-n}{1}\right| = \binom{n}{1} = \binom{n}{n-1} \tag{6.10}$$

and

$$\left|\binom{-n}{k+1}\right| = \binom{n+k-1}{k+1} = \binom{n+k-1}{n-1}. \tag{6.11}$$

For instance:

$$\begin{array}{ccccccc}
\left|\binom{-3}{1}\right| & = & \binom{3}{1} & = & \binom{3}{2} & = & 3 \\
\left|\binom{-3}{2}\right| & = & \binom{4}{2} & = & \binom{4}{2} & = & 6 \\
\left|\binom{-3}{3}\right| & = & \binom{5}{3} & = & \binom{5}{2} & = & 10 \\
\left|\binom{-3}{4}\right| & = & \binom{6}{4} & = & \binom{6}{2} & = & 15 \\
\left|\binom{-3}{5}\right| & = & \binom{7}{5} & = & \binom{7}{2} & = & 21 \\
\left|\binom{-3}{6}\right| & = & \binom{8}{6} & = & \binom{8}{2} & = & 28
\end{array}$$

This result may look a bit surprising at the first sight. But when we look at the formula that actually generates the value, it is obvious:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{6.12}$$

When we have a negative n, the falling factorial in the numerator is in fact a rising factorial with negative numbers:

$$-n^{\underline{k}} = -n \times -(n+1) \times \cdots \times -(n+k-1).$$

Each number in the product is one less than its predecessor, but, since the numbers are negative, the absolute value is greater than its predecessor. If we eliminate the minus signs, we obtain the rising factorial for n:

$$n^{\overline{k}} = n \times (n+1) \times \cdots \times (n+k-1),$$

which is just the falling factorial for $n+k-1$:

$$(n+k-1)^{\underline{k}} = (n+k-1) \times (n+k-2) \times \cdots \times n.$$

We can therefore conclude that

$$\left|\binom{-n}{k}\right| = \frac{(n+k-1)^{\underline{k}}}{k!} = \binom{n+k-1}{k}. \tag{6.13}$$

That $\binom{n+k-1}{k}$ also equals $\binom{n+k-1}{n-1}$ results from the fact that $n-1$ and k maintain the same distance from one of the sides of the triangle, *i.e.* from either $\binom{n+k-1}{0}$ or $\binom{n+k-1}{n+k-1}$. k is trivially k coefficients away from any $\binom{n}{0}$, whereas $n-1$ is $(n+k-1)-(n-1)$ away from $\binom{n+k-1}{n+k-1}$, which is $n+k-1-n+1=k$. This is just an implication of the triangle's symmetry.

Somewhat more difficult is to see the relation to Pascal's rule. In fact, we have never proven that Pascal's rule follows from the fraction $\frac{n^{\underline{k}}}{k!}$. If we can establish this relation, the backward rule will follow immediately. So, we definitely should try to prove Pascal's rule.

We want to establish that

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} \tag{6.14}$$

and do this directly using the definitions

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!} \tag{6.15}$$

$$\binom{n}{k-1} = \frac{n^{\underline{k-1}}}{(k-1)!} \tag{6.16}$$

Our claim is that

$$\binom{n+1}{k} = \frac{n^{\underline{k}}}{k!} + \frac{n^{\underline{k-1}}}{(k-1)!} \tag{6.17}$$

In other words, when we can deduce the left-hand side of this equation from the right-hand side, we are done. So let us just add the two fractions on the right-hand side. We first convert them to a common denominator. We can do this simply by multiplying the second fraction by k:

$$\frac{kn^{\underline{k-1}}}{k(k-1)! = k!}$$

We can join the two fractions with the common denominator and obtain a sum in the numerator:

$$\frac{n^{\underline{k}} + kn^{\underline{k-1}}}{k!}$$

If we extend the falling factorials in the numerator we get

$$n(n-1)\dots(n-k+1) + n(n-1)\dots(n-(k-1)+1)k.$$

The terms have the same number of factors, where the last factor in the first term is $n-k+1$ and the one in the second term is k. The factor before the last in the second term is $n-(k-1)+1$, which is $n-k+1+1 = n-k+2$, and this term is also the last but second factor in the first term. In other words, the factors of the terms are equal with the exception of the last factor. We can factor the equal factors out. If we do this stepwise, this looks like (using brackets to indicate what remains within the sum):

$$n[(n-1)(n-2)\dots(n-k+1)+(n-1)(n-2)\dots(n-k+2)k]$$
$$n(n-1)[(n-2)\dots(n-k+1)+(n-2)\dots(n-k+2)k]$$
$$\dots$$
$$n(n-1)\dots(n-k+2)[n-k+1+k].$$

The remaining sum can now be simplified: $n-k+1+k=n+1$ and with this we recognise in the whole expression the falling factorial of $n+1$. The whole fraction is now

$$\frac{(n+1)^{\underline{k}}}{k!},$$

which is the definition of the binomial coefficient $\binom{n+1}{k}$ and that concludes the proof. □

The proof establishes the relation between the definition of binomial coefficients and Pascal's rule. This spares us from going through the laborious task of establishing the relation expressed in equation 6.11 using only Pascal's rule.

We now switch to the first approach mentioned at the beginning of this section, *i.e.* trying to find a mathematical formalism for a practical problem. The practical problem is multiplication. We want to know what happens with negative as or bs in products of the form

$$(a+b)^n.$$

Negative ns are not too interesting here, since $(a+b)^{-n}$ is just the inverse of $(a+b)^n$, which is $\frac{1}{(a+b)^n}$, without any effect on the coefficients themselves.

We could, of course go on by trying out this formula with concrete numbers a and b. It appears much more promising, however, to choose a symbolic approach manipulating strings of the form `"a"` and `"b"`. The idea is to use string operations to simulate multiplication and addition. We do so in two steps: first we combine strings, then we simplify them in a way mimicking the rules of addition and multiplication. Since we want to see the differences between positive and negative coefficients, we need a means to negate strings simulating negative numbers. To this end, we define the simple data type

```
data Sym = P String | N String
  deriving (Eq, Show)
```

where, as you may have guessed, the P-constructor represents positive strings and the N-constructor represents negative strings. We now combine two symbols to simulate multiplication:

```
comb :: Sym → Sym → Sym
comb (P a) (P b) = P (a ++ b)
comb (P a) (N b) = N (a ++ b)
comb (N a) (P b) = N (a ++ b)
comb (N a) (N b) = P (a ++ b)
```

You probably realise the pattern we already used to define the number type *Zahl*: two numbers of equal signedness result in a positive number and two numbers of different signedness result in a negative number. Multiplication itself is just the concatenation of the two strings. *comb* (*P* `"a"`) (*P* `"b"`), hence, is *P* `"ab"`; *comb* (*N* `"a"`) (*P* `"b"`) is *N* `"ab"`.

We represent addition as lists of strings. We then can formulate the distributive law as

```
combN :: Sym → [Sym] → [Sym]
combN x = map (comb x)
```

where we multiply a number, x, with a sum by multiplying that number with each term of the sum, mapping the basic combinator operation *comb* on the list representing the sum. Based on *combN*, we can now define the multiplication of a sums by itself:

```
combine :: [Sym] → [Sym]
combine xs = concat [combN x xs | x ← xs]
```

Let us look at an example to get a grip on *combine*:

```
let a = P "a"
let b = P "b"
combine [a, b] = concat [combN x [a, b] | x ← [a, b]].
```

The list comprehension will first apply *combN a* $[a, b]$, resulting in $[aa, ab]$, and then it will apply *combN b* $[a, b]$ resulting in $[ba, bb]$. These two lists are then merged using *concat* resulting in $[aa, ab, ba, bb]$. We will later have to simplify this list, since, as we know, ab and ba are equal and can be written $2ab$. We come back to this immediately, but first we want to implement one more function, namely a combinator that applies *combine n* times:

```
combinator :: [Sym] → Natural → [Sym]
combinator _ 0 = []
combinator xs n = go xs (n − 1)
  where go ys 0 = ys
        go ys n = go (concat [combN x ys | x ← xs]) (n − 1)
```

Note that this function does not reuse *combine*, but implements it anew. The reason is that we do not want to combine the original input with itself, but with the result of the previous recursion. For instance, if $n = 3$, then we start with [*combN x* $[a, b]$ | $x \leftarrow [a, b]$] resulting in $[aa, ab, ba, bb]$. In the next round, we multiply this result by $[a, b]$: [*combN x* $[aa, ab, ba, bb]$ | $x \leftarrow [a, b]$] resulting in $[aaa, aab, aba, abb, baa, bab, bba, bbb]$

and corresponding to the expression $(a + b)^3$.

Another interesting aspect of *combinator* is the base case for $n = 0$, which is just defined as being the empty list. In this context, the empty list would, hence, represent the number 1, since 1, as we know, is the coefficient $\binom{0}{0}$.

Now we want to simplify results, such that $[aa, ab, ba, ba] = [aa, 2 * ab, ba]$. First, we need to express that `"ab"` and `"ba"` are the same thing. Therefore, we sort the string:

```
sortStr :: Sym → Sym
sortStr (P a) = P (sort a)
sortStr (N a) = N (sort a)
```

leading to a canonical form for strings. The call *sortStr* (*P* `"ba"`) would result in *P* `"ab"`.

To simplify a complete list, we compare all symbols in the lists and consolidate symbols with equal strings resulting in lists of the type (*Natural*, *Sym*), where the natural number counts the ocurrences of that symbol in the list:

```
simplify :: [Sym] → [(Natural, Sym)]
simplify xs = go (sort $ map sortStr xs) 1
  where go [] _ = []
        go [x] n = [(n, x)]
        go ((P x) : (P y) : zs) n | x ≡ y     = go ((P y) : zs) (n + 1)
                                  | otherwise = (n, P x) : go ((P y) : zs) 1
        go ((N x) : (N y) : zs) n | x ≡ y     = go ((N y) : zs) (n + 1)
                                  | otherwise = (n, N x) : go ((N y) : zs) 1
        go ((N x) : (P y) : zs) n | x ≡ y     = go zs 1
                                  | otherwise = (n, N x) : go ((P y) : zs) 1
        go ((P x) : (N y) : zs) n | x ≡ y     = go zs 1
                                  | otherwise = (n, P x) : go ((N y) : zs) 1
```

We start by first transforming all symbols into the canonical form sorting their strings. Then we sort the list of symbols itself. The idea is that all strings of the same kind are listed in a row, such that we can easily count them. But, of course, we do not want to have the negative and positive symbols separated, we want all symbols with the same string in a row, independently of these symbols being positive or negative. We have to implement this notion of comparison ignoring the sign. To this end, we make *Sym* an instance of *Ord*:

```
instance Ord Sym where
  compare (P a) (P b) = compare a b
  compare (P a) (N b) = compare a b
  compare (N a) (P b) = compare a b
  compare (N a) (N b) = compare a b
```

We now *go* through the list of symbols sorted in this sense of comparison and check on the first and the second of the remaining list on each step. If the signedness of first and second are equal and their strings are equal, we increment n and store (n, x), x the head of the list, whenever they differ starting the rest of the list with $n = 1$. Otherwise, when the signs are not equal, but the strings are, then we drop both symbols and continue with the rest of the list after the second.

Applied step for step on the list

$[aaa, aab, aba, abb, baa, bab, bba, bbb]$,

this would advance as follows. We first sort all strings resulting

$[aaa, aab, aab, abb, aab, abb, abb, bbb]$.

We next sort the symbols:

$[aaa, aab, aab, aab, abb, abb, abb, bbb]$.

We then weight according to the number of their appearance:

$[(1, aaa), (3, aab), (3, abb), (1, bbb)]$.

In this example, we ignore signedness, since all terms are positive anyway. The interesting thing thus is still to commence: the application to sums with negative terms. We do this with the simple function

$$\begin{aligned} &\mathit{powsum} :: [\mathit{Sym}] \to \mathit{Natural} \to [(\mathit{Natural}, \mathit{Sym})] \\ &\mathit{powsum}\ \mathit{xs} = \mathit{simplify} \circ \mathit{combinator}\ \mathit{xs} \end{aligned}$$

We now define two variables, a positive string a and a negative one b: **let** $a = P$ `"a"` and **let** $b = N$ `"b"` and just apply *powsum* with increasing ns:

powsum $[a, b]$ 1
$[(1, P$ `"a"`$), (1, N$ `"b"`$)]$

powsum $[a, b]$ 2
$[(1, P$ `"aa"`$), (2, N$ `"ab"`$), (1, P$ `"bb"`$)]$

powsum $[a, b]$ 3
$[(1, P$ `"aaa"`$), (3, N$ `"aab"`$), (3, P$ `"abb"`$), (1, N$ `"bbb"`$)]$

It appears that the absolute values of the coefficients do not change. We have, for instance, $\binom{3}{0} = 1$, $\binom{3}{1} = -3$, $\binom{3}{2} = 3$, $\binom{3}{3} = -1$. What, if we swap the negative sign: **let** $a = N$ `"a"` and **let** $b = P$ `"b"`?

powsum $[a, b]$ 1
$[(1, N$ `"a"`$), (1, P$ `"b"`$)]$

powsum $[a, b]$ 2
$[(1, P$ `"aa"`$), (2, N$ `"ab"`$), (1, P$ `"bb"`$)]$

```
powsum [a, b] 3
[(1, N "aaa"), (3, P "aab"), (3, N "abb"), (1, P "bbb")]
```

The result is just the same for even exponents. For odd exponents, the signs are just exchanged: $\binom{3}{0} = -1$, $\binom{3}{1} = 3$, $\binom{3}{2} = -3$, $\binom{3}{3} = 1$.

What if both terms are negative: **let** $a = N$ `"a"` and **let** $b = N$ `"b"`?

```
powsum [a, b] 1
[(1, N "a"), (1, N "b")]
```

```
powsum [a, b] 2
[(1, P "aa"), (2, P "ab"), (1, P "bb")]
```

```
powsum [a, b] 3
[(1, N "aaa"), (3, N "aab"), (3, N "abb"), (1, N "bbb")]
```

Now, wonder of wonders, even exponents lead to positive coefficients, while odd exponents lead to negative coefficients: $\binom{3}{0} = -1$, $\binom{3}{1} = -3$, $\binom{3}{2} = -3$, $\binom{3}{3} = -1$.

This result appears quite logical, since, with even exponents, we multiply an even number of negative factors, while, with odd exponents, we multiply an odd number of negative factors.

To confirm these results with more data, we define one more function to extract the coefficients and, this way, making the output more readable:

```
coeffs :: [Sym] → Natural → [Natural]
coeffs xs = map getCoeff ∘ powsum xs
```

where *getCoeff* is

```
getCoeff :: (Natural, Sym) → Natural
getCoeff (n, P _) = n
getCoeff (n, N _) = −n
```

We now map *coeffs* on the numbers $[0\mathinner{..}9]$ and, with a and b still both negative, see Pascal's triangle with signs alternating per row:

```
[]
[−1, −1]
[1, 2, 1]
[−1, −3, −3, −1]
[1, 4, 6, 4, 1]
[−1, −5, −10, −10, −5, −1]
[1, 6, 15, 20, 15, 6, 1]
[−1, −7, −21, −35, −35, −21, −7, −1]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
[−1, −9, −36, −84, −126, −126, −84, −36, −9, −1]
```

For one of the value a and b positive and the other negative, we see the coefficients in one row alternating in signedness. For a positive and b negative, we see:

$[\,]$
$[1, -1]$
$[1, -2, 1]$
$[1, -3, 3, -1]$
$[1, -4, 6, -4, 1]$
$[1, -5, 10, -10, 5, -1]$
$[1, -6, 15, -20, 15, -6, 1]$
$[1, -7, 21, -35, 35, -21, 7, -1]$
$[1, -8, 28, -56, 70, -56, 28, -8, 1]$
$[1, -9, 36, -84, 126, -126, 84, -36, 9, -1]$

The other way round, a negative and b positive, we see just the same triangle where, for odd exponents, the minus signs are swapped. The triangle, hence, is the same as the one before with each row reversed:

$[\,]$
$[-1, 1]$
$[1, -2, 1]$
$[-1, 3, -3, 1]$
$[1, -4, 6, -4, 1]$
$[-1, 5, -10, 10, -5, 1]$
$[1, -6, 15, -20, 15, -6, 1]$
$[-1, 7, -21, 35, -35, 21, -7, 1]$
$[1, -8, 28, -56, 70, -56, 28, -8, 1]$
$[-1, 9, -36, 84, -126, 126, -84, 36, -9, 1]$

6.4. $\mathbb{Q}$

We now turn our attention to fractions and start by implementing a rational data type:

```
data Ratio = Q Natural Natural
  deriving (Show, Eq)
```

A *Ratio* has a constructor Q that takes two natural numbers. The name of the constructor is derived from the symbol for the set of rational numbers $\mathbb{Q}$ that was introduced by Giuseppe Peano and stems from the Italian word *Quoziente*.

It would be nice of course to have a function that creates a rational in its canonical form, *i.e.* reduced to two natural numbers that are coprime to each other. This is done by *ratio*:

```
ratio :: Natural → Natural → Ratio
ratio _ 0 = error "division by zero"
ratio a b = reduce (Q a b)
```

for which we define the infix %:

```
infix %
(%) :: Natural → Natural → Ratio
(%) = ratio
```

so that we can create ratios with expressions like 5 % 2, 8 % 4 and so on. The function *reduce* called in *ratio* is defined as follows:

```
reduce :: Ratio → Ratio
reduce (Q _ 0) = error "division by zero"
reduce (Q 0 _) = Q 0 1
reduce (Q n d) = Q (n `div` gcd n d)
                   (d `div` gcd n d)
```

which reduces numerator and denominator to the quotient of the greatest common divisor of numerator and denominator. If numerator and denominator are coprime, the *gcd* is just 1 and the numbers are not changed at all.

Useful would be to have access functions for numerator and denominator. We define them straight forward as

```
numerator :: Ratio → Natural
numerator (Q n _) = n
denominator :: Ratio → Natural
denominator (Q _ d) = d
```

We now make *Ratio* an instance of *Ord*:

```
instance Ord Ratio where
  compare x@(Q nx dx) y@(Q ny dy) | dx ≡ dy   = compare nx ny
                                  | otherwise = let (x′, y′) = unify x y
                                                in compare x′ y′
```

If the denominators are equal, we just compare the numerators, *i.e.* $\frac{1}{5} < \frac{2}{5} < \frac{3}{5}$ and so on. Otherwise, if the denominators differ, we must convert the fractions to a common denominator before we actually can compare them. This is done using *unify*:

```
unify :: Ratio → Ratio → (Ratio, Ratio)
unify (Q nx dx) (Q ny dy) = (Q (nx ∗ (lcm dx dy) `div` dx) (lcm dx dy),
                             Q (ny ∗ (lcm dx dy) `div` dy) (lcm dy dx))
```

This is the implementation of the logic already described before: we convert a fraction to the common denominator defined by the *lcm* of both denominators and multiply the numerators by the number we would have to multiply the denominator by to get the *lcm*, which trivially is the *lcm* divided by the denominator: $lcm(dx, dy) = dx \times \frac{lcm(dx,dy)}{dx}$.

This may appear a bit complicated, but it is much faster, whenever the denominators are not coprime to each other.

The next step is to make *Ratio* an instance of *Enum*:

```
instance Enum Ratio where
  toEnum i = Q (toEnum i) (toEnum 1)
  fromEnum (Q n d) = fromEnum (n ‘div‘ d)
```

which implies a conversion from and to an integer type. A plain integral number i is converted to a *Ratio* using the denominator 1. For the backward conversion, *div* is used leading to the loss of precision if the denominator does not divide the numerator.

Now we come to the heart of the data type making it instance of *Num*:

```
instance Num Ratio where
  x@(Q nx dx) + y@(Q ny dy) | dx ≡ dy   = (nx + ny) % dx
                            | otherwise = let (x′, y′) = unify x y
                                          in (x′ + y′)
  x@(Q nx dx) − y@(Q ny dy) | x ≡ y = Q 0 1
                            | x > y ∧ dx ≡ dy = (nx − ny) % dx
                            | x > y = let (x′, y′) = unify x y
                                      in x′ − y′
                            | otherwise = error "Subtraction beyond zero!"
  (Q nx dx) ∗ (Q ny dy) = (nx ∗ ny) % (dx ∗ dy)
  negate a = a
  abs a = a
  signum (Q 0 _) = 0
  signum (Q _ _) = 1
  fromInteger i = Q (fromIntegral i) 1
```

We add two fraction with the same denominator by reducing the result of $\frac{nx+ny}{d}$. If we add two fractions in canonical form, such as $\frac{1}{9}$ and $\frac{5}{9}$, we may arrive at a result that is not in canonical form like, in this example, $\frac{6}{9}$, which should be reduced to $\frac{2}{3}$. Otherwise, if the denominators differ, we first convert the fractions to a common denominator before we add them.

Since we have defined *Ratio* as a fraction of two natural numbers (and not of two integers), we have to be careful with subtraction. If the two fractions are equal, the result is zero, which is represented as $\frac{0}{1}$. If $x > y$, we use the same strategy as with addition. Otherwise, if $y > x$, subtraction is undefined.

Multiplication is easy: we just reduce the result of multiplying the two numerators and denominators by each other. The other functions do not add anything new. We just define *negate*, *abs* and *signum* as we have done before for plain natural numbers and we define the conversion from integer as we have done for *Enum* already.

The next step, however, is unique: we define *Ratio* as an instance of *Fractional*. The core of this is to define a division function and do so defining division as the inverse of multiplication:

```
rdiv :: Ratio → Ratio → Ratio
rdiv (Q nx dx) (Q ny dy) = (Q nx dx) * (Q dy ny)
```

The division of a fraction $\frac{nx}{dx}$ by another $\frac{ny}{dy}$ is just the multiplication of that fraction with the inverse of the second one, which is $\frac{dy}{ny}$. The complete implementation of the *Fractional* type then is

```
instance Fractional Ratio where
  (/)            = rdiv
  fromRational r = Q (fromIntegral $ R.numerator r)
                     (fromIntegral $ R.denominator r)
```

This is not a complete definition of $\mathbb{Q}$, however. $\mathbb{Q}$ is usually defined on top of the integers rather than on top of natural numbers. So, our data type should be signed. That, however, is quite easy to achieve:

```
type Quoz = Signed Ratio
```

6.5. Zeno's Paradox

The ancient Greek philosopher Zeno of Elea, who lived in the 5^{th} century BC, devised a number of paradoxes that came upon us indirectly through the work of Aristotle and its commentators. Zeno designed the paradoxes to defend the philosophy of the *One* developed by Zeno's teacher Parmenides. According to this philosophy everything is One, undivisible, motionless, eternal, everywhere and nowhere. That we actually see motion, distinguish and divide things around us, that everything "in this world" is volatile and that everything has its place, is either here or there, is, according to Parmenides, just an illusion.

Plato discusses the philosophy of Parmenides in one of his most intriguing dialogs, "Parmenides", where young Socrates, Zeno and Parmenides himself analyse contradictions that arise both in Plato's *theory of forms* as well as in Parmenides' theory of the One. The Parmenides dialog had a deep influence on European philosophy and religion. It was the main inspiration for the late-ancient *neoplatonism* and, for many centuries, it shaped the interpretation of ancient philosophy by medieval thinkers. Scholars today, however, are not so sure anymore what the meaning of this dialog may be. Some see in it a critical discussion of the theory of forms, others hold it is a collection of exercises for students of Plato's academy, and again others consider the dialog as highly ironic, actually criticising Parmenides and other philosophers for using terms that they know from everyday life in a context where the ideas associated with this terms do not hold anymore – very similar to the therapeutic approach of Ludwig Wittgenstein.

It is tempting to relate the philosophy of the One with the philosophical worldview of mathematical platonism. Constructivists would see the world as dynamic, as a chaotic process without meaning in itself or, pessimistically, as a thermodynamic process that tends to entropy. It is an effort of human beings to create order in this dynamic and perhaps chaotic world. Therefore, prime numbers – or any other mathematical object – do not exist, we define them and we have to invest energy to construct them. By contrast, mathematical platonists would hold that there is a static eternal structure that does not change at all. The mathematical objects are out there, perhaps like in a gigantic lattice behind the universe. It would then be absurd to say that we construct prime numbers. We find them travelling along the eternal metaphysical structure that is behind of what we can perceive directly with our senses.

Be that as it may, we are here much more interested in the math in Zeno's paradoxes. The most famous one is the race of Achilles and the tortoise. Achilles gives the tortoise a lead of, say, hundred meters. The question now is when Achilles will actually catch up with the tortoise. Zeno says: never, for it is impossible. To catch up, he must reach a point where the tortoise has been shortly before. But when he gets there, the tortoise is already ahead. Perhaps just a few metres, but definitely ahead. So, again, to reach that point, Achilles will need some time. When he reaches the point where the tortoise has been a second before, the same is already a bit further. To reach that point, Achilles again needs some time and in this time the tortoise again makes some progress and so it goes on and on.

A more concrete version of this paradox is given in the so called *Dichotomy* paradox. It states that, in general, it is impossible to move from A to B. Since, to do so, one has first to make half of the way arriving at a point C. To move from C to B, one now has to first make half of the way arriving at a point D. To move from D to B, one now has to first make half of the way arriving at yet another point and so on and so on.

This paradox appears to be at odds with what we observe in the physical world where it indeed appears to be possible to move from A to B quite easily. The paradox, however, draws our attention to the fact that, between any two rational numbers, there are infinitely many other rational numbers. Between 0 and 1, for instance, there is $\frac{1}{2}$. Between 0 and $\frac{1}{2}$, there is $\frac{1}{4}$. Between 0 and $\frac{1}{4}$, there is $\frac{1}{8}$. Between 0 and $\frac{1}{8}$, there is $\frac{1}{16}$ and, in general, between 0 and any number of the form $\frac{1}{2^k}$, there is a number $\frac{1}{2^{k+1}}$.

So, following these points as in Zeno's paradox, how close to B would we get after k steps? The problem can be represented as a sum of the form

$$\sum_{i=1}^{k}\frac{1}{2^i}$$

After $k = 1$ step, we would be $\frac{1}{2}$ of the way away from B. After $k = 2$ steps, the distance would be

$$\frac{1}{2}+\frac{1}{2^2}.$$

We convert the fractions to a common denominator multiplying the first fraction by 2 and arrive at

$$\frac{2+1}{4}=\frac{3}{4}.$$

For $k = 3$ steps, we have

$$\frac{3}{4}+\frac{1}{2^3}=\frac{6+1}{8}=\frac{7}{8}.$$

These tests suggest the general formula

$$\sum_{i=1}^{k}\frac{1}{2^i}=\frac{2^k-1}{2^k}. \tag{6.18}$$

This equation cries out for an induction proof. Any of the examples above serves as base case. We then have to prove that

$$\frac{2^k-1}{2^k}+\frac{1}{2^{k+1}}=\frac{2^{k+1}-1}{2^{k+1}}. \tag{6.19}$$

We convert the fractions to a common denominator multiplying the first fraction by 2:

$$\frac{2(2^k-1)+1}{2^{k+1}}.$$

We simplify the numerator: $2 \times 2^k = 2^{k+1}$ and $2 \times (-1) = -2$; we, hence, have in the numerator $2^{k+1} - 2 + 1$, which can be simplified to $2^{k+1} - 1$ and leads to the desired result

$$\frac{2^{k+1}-1}{2^{k+1}}. \quad \square$$

That was easy! Can we generalise the result for any denominator n, such that

$$\frac{1}{n^k}+\frac{1}{n^{k+1}}=\frac{n^{k+1}-1}{n^{k+1}}? \tag{6.20}$$

If we went a third of the way on each step instead of half of it, we had $\frac{1}{3^k} + \frac{1}{3^{k+1}}$, for instance, $\frac{1}{3} + \frac{1}{9}$. We convert the fraction to a common denominator multiplying the first by 3: $\frac{3+1}{9} = \frac{4}{9}$. So, equation 6.20 seems to be wrong. The nice and clean result with the denominator 2 appears to be one of those deceptions that are so common for small numbers, which often behave very differently from greater numbers.

But let us stop moaning. What actually is the rule for $n = 3$? After the next step, we would have

$$\frac{4}{9} + \frac{1}{27}.$$

We multiply the first fraction by 3 and have

$$\frac{12+1}{27} = \frac{13}{27}.$$

For $k = 4$, we would have

$$\frac{13}{27} + \frac{1}{81} = \frac{39+1}{81} = \frac{40}{81}.$$

The experiments this time suggest the rule

$$\sum_{i=1}^{k} \frac{1}{3^i} = \frac{(3^k - 1)/2}{3^k}. \tag{6.21}$$

We prove again by induction with any of the examples serving as base case. We have to prove that

$$\frac{(3^k - 1)/2}{3^k} + \frac{1}{3^{k+1}} = \frac{(3^{k+1} - 1)/2}{3^{k+1}}. \tag{6.22}$$

We multiply the first fraction by 3 in numerator and denominator and get in the numerator $\frac{3(3^k-1)}{2} = \frac{3^{k+1}-3}{2}$. We can now add the two fractions:

$$\frac{(3^{k+1} - 3)/2 + 1}{3^{k+1}}.$$

To add 1 to the fraction in the numerator we have to convert 1 to a fraction with the denominator 2, which, of course, is $\frac{2}{2}$. We, hence, have in the numerator $\frac{3^{k+1}-3+2}{2}$ and this leads to the desired result:

$$\frac{(3^{k+1}-1)/2}{3^{k+1}}. \quad \square \tag{6.23}$$

Before we dare to make a new conjecture based on equations 6.18 and 6.21, let us collect some more data. Since $n = 4$ is closely related to $n = 2$, we will immediately go to $n = 5$. For $k = 2$ we have

$$\frac{1}{5} + \frac{1}{25} = \frac{5+1}{25} = \frac{6}{25}.$$

For $k = 3$ we have

$$\frac{6}{25} + \frac{1}{125} = \frac{30+1}{125} = \frac{31}{125}.$$

For $k = 4$ we have

$$\frac{31}{125} + \frac{1}{625} = \frac{155+1}{625} = \frac{156}{625}.$$

In these examples, we see the relation

$$\sum_{i=1}^{k} \frac{1}{5^i} = \frac{(5^k-1)/4}{5^k}. \tag{6.24}$$

We prove easily by induction using any of the examples as base case. We have to show that

$$\frac{(5^k-1)/4}{5^k} + \frac{1}{5^{k+1}} = \frac{(5^{k+1}-1)/4}{5^{k+1}}. \tag{6.25}$$

We multiply the first fraction by 5, yielding the numerator $\frac{5^{k+1}-5}{4}$ and, when adding 1, we get $\frac{5^{k+1}-5}{4} + \frac{4}{4}$, which, of course, leads to the desired result. $\square$

To summarise: with $n = 2$, we see $\frac{n^k-1}{n^k}$; with $n = 3$, we see $\frac{(n^k-1)/2}{n^k}$; with $n = 5$, we see $\frac{(n^k-1)/4}{n^k}$. This suggests the general form

$$\sum_{i=1}^{k} \frac{1}{n^i} = \frac{(n^k-1)/(n-1)}{n^k}, \tag{6.26}$$

which would nicely explain why we overlooked the division in the numerator for the case $n = 2$, since, here, $n - 1 = 1$ and anything divided by 1 is just that anything.

It, again, does not appear to be too difficult to prove the result. We have a lot of base cases already and now want to prove that

$$\frac{(n^k - 1)/(n-1)}{n^k} + \frac{1}{n^{k+1}} = \frac{(n^{k+1} - 1)/(n-1)}{n^{k+1}}. \tag{6.27}$$

We multiply the first fraction by n in numerator and denominator and get in the numerator

$$\frac{n(n^k - 1)}{n-1} = \frac{n^{k+1} - n}{n-1}.$$

We now add 1 represented as the fraction $\frac{n-1}{n-1}$:

$$\frac{n^{k+1} - n}{n-1} + \frac{n-1}{n-1},$$

leading to

$$\frac{n^{k+1} - n + n - 1}{n-1} = \frac{n^{k+1} - 1}{n-1},$$

which is the desired result

$$\frac{(n^{k+1} - 1)/(n-1)}{n^{k+1}}. \quad \square \tag{6.28}$$

6.6. Systems of Linear Equations

Systems of linear equations provide an excellent topic to get familar with structures that we will need a lot in algebra, namely *matrices*. Before we get there, we look at linear equations as such. Linear equations and the systems made of them belong to the oldest topics studied in algebra. There is a rich body of knowledge in Chinese and Indian books dating back to antiquity and early middle ages (in terms of European history). The famous "Nine Chapters of Mathematical Art", for instance, dates back to 179 AD. It contains advanced algorithms to solve systems of linear equations that were formulated in Europe only in the 19^{th} century.

This knowledge was brought to Europe through Arab and Persian scholars, most famously perhaps al-Hwarizmi, called Algoritmi in medieval Europe, and his book "Compendium on Calculation and Balancing", whose original title contains the word "al-gabr", which was latinised as *algebra*.

In this tradition, systems of linear equations were often worded in terms of *bird problems*. Bird problems are centered around the question of how many of n different kinds of birds can be bought for a specific amount of money. A typical problem is to buy 100 birds for 100 drachme. There are ducks, chickens and sparrows. For 1 drachme, you can either buy one chicken or 20 sparrows; for 5 drachme, you get a duck. This translates into the simple system of equations

$$\begin{array}{lcl} x + y + z & = & 100 \\ 5x + \frac{1}{20}y + z & = & 100 \end{array} \tag{6.29}$$

The first equation states that the sum of the number of birds shall be 100; the second equation states that the sum of the price of the birds shall be 100 drachme.

One way to solve such equations is to *eliminate* one of the variables. In the given system, we can solve for z in both equations:

$$z = 100 - x - y \tag{6.30}$$

and

$$z = 100 - 5x - \frac{1}{20}y. \tag{6.31}$$

We set the right-hand sides of the equations equal, subtract 100 and bring x to the left side of the equation and y to the right side. We get:

$$4x = \frac{19}{20}y \tag{6.32}$$

and divide by 4:

$$x = \frac{19}{80}y. \tag{6.33}$$

From here, we easily find a solution by assuming that $y = 80$, $x = 19$ and, in consequence, $z = 1$. The original equations with the variables substituted, then, read

$$\begin{array}{lcl} 19 + 80 + 1 & = & 100 \\ 5 \times 19 + \frac{1}{20} \times 80 + 1 & = & 100, \end{array} \tag{6.34}$$

which, as you can easily convince yourself, is correct in both cases.

The final step in the derivation was a mere guess based on the fact that we expected integer numbers as results in one of the equations. Without that restriction, *i.e.* when we define the system over the field of rational numbers, would there be a way to solve any such system? It turns out, there is. Furthermore, that algorithm is guaranteed to find a single solution to any well-defined system.

You might remember a similar claim that we proved for a special kind of systems in the previous chapter, namely the Chinese Remainder Theorem. Indeed, Chinese remainders are just a special case of linear equations in a finite field of modular arithmetic. For the general case, which includes infinite fields, such as the rational numbers, we have to restrict the claim adding the constraint that the system must be *well-defined*.

By this, we mean that the system is *consistent* and contains the same number of *independent* equations and unknowns; for the bird problem above this was not the case, since there were only two equations for three unknowns (x, y and z). However, the bird problem was restricted to integers, and we were able to guess the result after some steps.

Have a look at the following system:

$$\begin{array}{rcrcl} x & + & y & = & 1 \\ 2x & + & 2y & = & 2 \end{array} \tag{6.35}$$

There are two unknowns, x and y, and two equations. Unfortunately, the two equations are not independent, since the second equation is equivalent to the first, *i.e.* it is just the first equation scaled up. Indeed, whenever one equation can be derived from the others by algebraic means, it is not independent and, hence, does not add new information to the system. A somewhat more subtle example of a system with a dependent equation is

$$\begin{array}{rcrcrcr} x & - & 2y & + & z & = & -1 \\ 3x & + & 5y & + & z & = & 8 \\ 4x & + & 3y & + & 2z & = & 7. \end{array} \tag{6.36}$$

Here, the third equation is the sum of equations 1 and 2, so it does not add new information.

Systems of equations that have more unknowns than independent equations are called *underdetermined*. They usually have no or infinitely many solutions. If a system has more equations than unknowns, it is *overdetermined* and, usually, has no solution. The system is then *inconsistent*, *i.e.* it contains a contradiction. An inconsistent system is, for instance

$$\begin{array}{rcrcrcr} x & - & 2y & + & z & = & -1\\ 3x & + & 5y & + & z & = & 8\\ 4x & + & 3y & + & 2z & = & 5. \end{array} \tag{6.37}$$

The sum of the left-hand side of equations 1 and 2 results in the left-hand side of equation 3. The right-hand side of equation 3, however, is not the sum of the right-hand side of equations 1 and 2. Any try of to solve this system will lead to a contradiction of the form $1 = 0$.

A consistent system, however, that has the same number of independent equations and unknowns has, within a field, always a unique solution and there is an algorithm that finds this solution. But before we present and implement the algorithm as such, we will look at the ideas, on which it is based.

The first approach is *elimination*. The idea is to solve one equation for one of the variables and then to substitute that variable in the other equations by the result. A concrete example:

$$\begin{array}{rcrcrcr} x & + & 3y & - & 2z & = & 5\\ 3x & + & 5y & + & 6z & = & 7\\ 2x & + & 4y & + & 3z & = & 8. \end{array} \tag{6.38}$$

We solve the first equation for x. We just subtract $3y$ from and add $2z$ to both side to obtain

$$x = 5 - 3y + 2z. \tag{6.39}$$

We substitute this result for x in the other equations and obtain:

$$\begin{array}{rcrcrcr} 3(5-3y+2z) & + & 5y & + & 6z & = & 7\\ 2(5-3y+2z) & + & 4y & + & 3z & = & 8, \end{array} \tag{6.40}$$

which, after simplication and bringing the constant numbers to the right-hand side, translates to

$$\begin{array}{rcrcr} -4y & + & 12z & = & -8\\ -2y & + & 7z & = & -2. \end{array} \tag{6.41}$$

Now we repeat the process, solving the first of these equations for y, which yields $-4y = -12z - 8$ and, after dividing both sides by -4, $y = 3z + 2$. We then substitute y into the third equation yielding $-2(3z+2)+7z = -2$. Simplifying again leads to $z-4 = -2$ and, after adding 4 to both sides, $z = 2$.

Now, we just go backwards, first substituting z in the equation solved for y leading to

$$y = 3 \times 2 + 2 = 8 \tag{6.42}$$

and, second, substituting $z = 2$ and $y = 8$ in the first equation solved for x:

$$x = 5 - 3 \times 8 + 2 \times 2 = -19 + 4 = -15. \tag{6.43}$$

The complete result, hence, is

$$x = -15, y = 8, z = 2.$$

Notice that the approach aims to subsequently *eliminate* variables from the equations. This way, we simplify a system with n equations and unknowns to a system with $n - 1$ equations and unknowns and, then, we just repeat until we are left with one equation with one unknown.

We can reach this goal in a more direct manner by adding (or subtracting) one equation to (or from) the other such that one of the unknowns disappears, *i.e.* reduces to zero. Usually, we have to scale one of the equations to achieve this.

When we look at the previous system once again

$$\begin{array}{rcrcrcr} x & + & 3y & - & 2z & = & 5 \\ 3x & + & 5y & + & 6z & = & 7 \\ 2x & + & 4y & + & 3z & = & 8, \end{array} \tag{6.44}$$

we see that, if we scale the first equation by factor 3 and add it to the second equation, z would fall away:

$$\begin{array}{crcrcrcr} & 3x & + & 9y & - & 6z & = & 15 \\ + & 3x & + & 5y & + & 6z & = & 7 \\ = & 6x & + & 14y & & & = & 22. \end{array} \tag{6.45}$$

Likewise, we can scale the third equation by factor 2 and subtract it from the second equation:

$$\begin{array}{crcrcrcr} & 3x & + & 5y & + & 6z & = & 7 \\ - & 4x & + & 8y & + & 6z & = & 16 \\ = & -x & - & 3y & & & = & -9. \end{array} \tag{6.46}$$

This way, we obtain two equations with two unknowns. We can eliminate one more unknown by scaling the second of these new equations by factor 6 and add it to the first one:

$$\begin{array}{rrcrcr} & 6x & + & 14y & = & 22 \\ + & -6x & - & 18y & = & -54 \\ = & & & -4y & = & -32. \end{array} \tag{6.47}$$

When we divide both sides of the result by -4, we get $y = 8$, which is the same result we saw before with the elimination method.

We now can go on and eliminate other unknowns by scaling and adding. We should not be frightened to use fractions, when solving equations in a field. We can, for instance, isolate x by scaling the resulting equation 6.46 by the factor $\frac{14}{3}$ and add it to equation 6.45:

$$\begin{array}{rrcrcr} & 6x & + & 14y & = & 22 \\ + & -\frac{14}{3}x & - & 14y & = & -42 \\ = & \frac{4}{3}x & & & = & -20. \end{array} \tag{6.48}$$

After multiplying by 3 and dividing by 4 on both sides, we get $x = -15$, as before.

The generic algorithm is based on these principles of scaling and adding as well as elimination, but does so in a systematic way. In our manual process, we took decisions on which equation to solve and on which equations to add to or subtract from which other. Those decisions were driven by human motives, for instance, to avoid fractions whenever possible. For a systematic algorithm executed on a machine, such considerations are irrelevant. The machine has no peference for integers over fractions.

The algorithm is called *Gaussian elimination*, although it is known to Chinese and Indian mathematicians since late antiquity. We will here discuss the basic form of this algorithm. There is a more advanced form, called *Gauss-Jordan algorithm*, at which we look later. Interesting, however, is the second eponym of the algorithm, Wilhelm Jordan (1842 – 1899), a German geodesist. This underlines the fact that this method – as well as many other methods from linear algebra – has its roots in applied science rather than in pure mathematics.

Both algorithms are based on a data structure of fundamental importance in linear algebra, the *matrix*. We, here, introduce matrices as a mere tool that helps us doing calculations. In algebra, however, matrices are studied as a topic in itself.

Anyway, what is a matrix in the first place? Well, “matrix” is basically a fancy name for what we all know as “table”. A matrix consists of rows and columns that are identified by a pair of indices (i, j), where i usually refers to the row and j to the column.

Here we use matrices to represent systems of equations. Each row contains one equation.

Each column contains one coefficient, *i.e.* the numbers before the unknowns and, in the last column, we have the constant value on the right-hand side of the equations (this is often called an *augmented matrix*). Our equation above can be represented in matrix form as:

$$\begin{pmatrix} 1 & 3 & -2 & 5 \\ 3 & 5 & 6 & 7 \\ 2 & 4 & 3 & 8 \end{pmatrix}$$

In Haskell, we can define a matrix as a list of lists, where the inner lists represent rows, for instance:

```
data Matrix a = M [[a]]
  deriving (Show, Eq)
```

We can create a matrix for our system by

```
mysystem :: Matrix [Natural]
mysystem = let e1 = [1, 3, −2, 5]
               e2 = [3, 5, 6, 7]
               e3 = [2, 4, 3, 8]
           in  M [e1, e2, e3]
```

The following functions yield the rows and, respectively, the columns of the matrix:

```
rows :: Matrix a → [[a]]
rows (M rs) = rs

cols :: Matrix a → [[a]]
cols (M rs) = go rs
  where go [] = []
        go rs | null (head rs) = []
              | otherwise =
                heads rs : go (tails rs)

heads :: [[a]] → [a]
heads zs = [head z | z ← zs, ¬ (null z)]

tails :: [[a]] → [[a]]
tails = map tail z
```

Obtaining the rows is trivial: the function just returns the list of lists. Columns are bit more difficult. We recursively return the list of the heads of the inner lists, reducing these lists per step to their *tail*s until the lists are empty. This condition is checked on the first inner list. Since, in a matrix, all rows need to have the same size, the first list can act as a model for all lists.

Here are two helper functions to compute the length of one row in the matrix and to compute the length of one column in the matrix:

```
colen :: [[a]] → Int
colen = length

rowlen :: [[a]] → Int
rowlen []      = 0
rowlen [x: _] = length x

columnLength :: Matrix a → Int
columnLength (M ms) = colen ms

rowLength :: Matrix a → Int
rowLength (M ms) = rowlen ms
```

The column length is equivalent to the number of rows in the matrix; the row length is the length of the first row. Again, in a matrix, all rows shall have the same length; the first row, hence, serves as a pattern for the other rows.

Gaussian elimination consists of two steps (one of the improvements of Gauss-Jordan is that it consists of only one step, but applies this step with more consequence). The first step brings the matrix into a special form, often called *echelon* form. In this form, the matrix contains a triangle of zeros in the lower-left corner like this:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ 0 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 0 & 0 & a_{2,2} & a_{2,3} & a_{2,4} \\ 0 & 0 & 0 & a_{3,3} & a_{3,4} \end{pmatrix}.$$

The echelon form of our matrix is as follows:

$$\begin{pmatrix} 1 & 3 & -2 & 5 \\ 0 & 4 & -12 & 8 \\ 0 & 0 & 4 & 8 \end{pmatrix}$$

The echelon form corresponds to a system of equations where the last equation has been reduced to one unkown; the last but one to two unknowns and so one until the first that remains in its original form.

The second step consists in eliminating and backsubstituting coefficients remaining in the matrix. But let us first look at how to create the echelon form. In Haskell this may be implemented as follows:[1]

---

[1] This code is based on Matrix.hs, part of the Hugs system

```
echelon :: (Eq a, Num a) ⇒ Matrix a → Matrix a
echelon (M ms) = M (go ms)
  where go :: (Eq a, Num a) ⇒ [[a]] → [[a]]
        go rs | null rs ∨
                  null (head rs) = rs
              | null rs2         = map (0:) (go (map tail rs))
              | otherwise        = piv : map (0:) (go rs′)
           where rs′             = map (adjustWith piv) (rs1 ++ rs3)
                 (rs1, rs2)      = span (λ(n: _) → n ≡ 0) rs
                 (piv : rs3)     = rs2
adjustWith :: (Num a) ⇒ [a] → [a] → [a]
adjustWith (m : ms) (n : ns) = zipWith (−) (map (n∗) ms)
                                           (map (m∗) ns)
```

We first look at *adjustWith*. This function takes two lists (of equal length), drops the first elements, scales each of the lists multiplying by the first element of the respective other list and zips the result together by subtracting the corresponding elements. Note that, if we not dropped the first elements, they would be multiplied by the first element of the respective other list; in consequence, both lists would begin with *nm*. Subracting one list from the other would result in a list with a leading zero. The function, instead, just drops the leading element.

Now let us look at how *adjustWith* is used in *echelon*. The main work in *echelon* is done in the local function *go*. This function has two base cases:

1. If the input matrix *rs* is null (it contains no rows) or if its first element is null (it contains only empty rows), the input is already in echelon form and we give it back as is.
2. We look at the local variable *rs2*. This variable is generated as the second element of a tuple resulting from *span* $(\lambda(n{:}\ \_) \rightarrow n \equiv 0)$, i.e. *rs1* will contain the rows with leading zeros and *rs2* will contain those without leading zeros. If *rs2* is the empty list, all rows in *rs* have at least one leading zero; we, therefore, ignore this step and continue with the tail of all rows, adding the zero that we ignored here to the final result again.

Now, in the *otherwise* branch, we use the *adjustWith* function. It is used to generate the local variable *rs′*. Look at how this variable is generated: (*adjustWith piv*) is mapped on the concatenation *rs1* ++ *rs3*. We already know the variable *rs1*: it contains the rows of *rs* with leading zeros. The second list, *rs3*, is created from *rs2* as (*piv* : *rs3*). The pivot (*piv*), hence, is the first row without leading zero and *rs3* consists of all other rows. In other words: we use one row (the pivot) to eliminate one variable from all rows. From here, it is simple: we just apply *go* once again on the result *rs′* until one of the base cases applies. On each step, we insert zero as head to all rows in the result matrix and, finally, add one more row: the pivot that now contains on more column with a value

$\neq 0$ than the rows in the result matrix. The code looks a bit scary on the first sight, but, after going through it step by step, it turns out to be quite simple. But let us go through an example: in the first instance of *go*, we compute

$(\mathit{rs1}, \mathit{rs2}) = ([\,], m)$, where m contains all lines of the matrix;
$(\mathit{piv}, \mathit{rs3}) = ([1, 3, -2, 5], \mathit{rs3})$, where *rs3* contains the last two lines.

For *adjustWith piv*, we compute, for the first line of *rs3*:

$$\begin{array}{crrrr} & 3 & 9 & -6 & 15 \\ - & 3 & 5 & 6 & 7 \\ = & 0 & 4 & -12 & 8 \end{array} \tag{6.49}$$

and for the second:

$$\begin{array}{crrrr} & 2 & 6 & -4 & 10 \\ - & 2 & 4 & 3 & 8 \\ = & 0 & 2 & -7 & 2 \end{array} \tag{6.50}$$

.

With these results, we repeat the process computing

$(\mathit{rs1}, \mathit{rs2}) = ([\,], m)$, where m now contains the two results computed above;
$(\mathit{piv}, \mathit{rs3}) = ([4, -12, 8], [[2, -7, 2]])$.

For *adjustWith piv*, we compute:

$$\begin{array}{crrr} & 8 & -24 & 16 \\ - & 8 & -28 & 8 \\ = & 0 & 4 & 8 \end{array} \tag{6.51}$$

Now, going back, we add heading zeros to the rows and, per recursion, the pivot resulting in the matrix:

$$\begin{pmatrix} 1 & 3 & -2 & 5 \\ 0 & 4 & -12 & 8 \\ 0 & 0 & 4 & 8 \end{pmatrix}$$

It should be clear, by the way, that *echelon* just applies the second method we discussed above: it systematically scales equations (in *adjustWith*) and subtracts them from each other. Now you may guess that the second step of the algorithm applies the first method, *i.e.* eliminating variables by solving and back-substituting – and you are right:[2]

---

[2] This code is based on *Haskell Road*

```
backsub :: Matrix Zahl → [Quoz]
backsub (M ms) = go ms []
  where go [] rs = rs
        go xs rs = go xs′ (p : rs)
          where a       = (last xs) !! ((rowlen xs) − 2)
                c       = (last xs) !! ((rowlen xs) − 1)
                p       = c % a
                (M xs′) = eliminate p $ M (init xs)
eliminate :: Quoz → Matrix Zahl → Matrix Zahl
eliminate r (M ms) = M (map (simplify n d) ms)
  where n = numerator   r
        d = denominator r
        simplify n d row = init (init row′) ⧺ [d ∗ lr − al ∗ n]
          where lr   = last row
                al   = last (init row)
                row′ = map (∗d) row
```

Note that, for sake of the topic of this section, we have adapted the code to a specific data type. The function *backsub* expects a system with integer coefficients and presents a result of rational numbers.

In *backsub*, we call the local function *go*, which receives the input matrix and an empty result set. When the input matrix is exhausted, we yield the result set. Otherwise, we create the local variables xs' and p. The latter is a rational number generated by dividing the last element of the last row by the penultimate element of that same row. This number, the quotient of the last and the last but one element of the last row, is the first element of the result set.

What does that mean? Well, look at the last line of the matrix. It reads $0, 0, 4, 8$. That is: it contains only two elements. The penultimate element, 4, is the coefficient of the last unknown, z, while the last element, 8, is the constant value on the right-hand side of this equation with one unknown. The last row can thus be rephrased as:

$$4z = 8.$$

That we divide the last element by the last but one corresponds to the simple manipulation that divides both sides of the equation by 4, *i.e.*

$$z = \frac{8}{4} = 2.$$

The other variable xs' is generated by eliminating p from the other lines. Eliminating works as follows: We first multiply all elements in every row by the denominator of

p. This corresponds to the following operation; the last but one row of the matrix, for instance, is

$$4y - 12z = 8.$$

When we substitute z by $\frac{8}{4}$ (which, of course, is 2, but let us look at the fraction), we get:

$$4y - \frac{12 \times 8}{4} = 8.$$

We now get rid of the denominator, by multiplying both sides by 4:

$$16y - 12 \times 8 = 32.$$

In the code above, we start by performing this second step, *i.e.* multiplying by the denominator. Note, however, that we later continue to compute with the last and the last but one element of row, not of row'. In other words, we multiply the denominator only by the elements that precede the penultimate and leave the last two elements as they are.

We then take the last two elements, multiply the last one by the denominator and the penultimate one by the numerator and subtract the latter from the former. That is, we get rid of the denominator, apply multiplication of the numerator to the value that represents z and subtract it from both sides. In abstract algebraic notation, that would look like:

$$ay + bz = c.$$

We know that $z = \frac{n}{d}$, so we can substitute the second term for $\frac{bn}{d}$. We multiply by d and get:

$$ady + bn = cd.$$

Now, we subtract bn from both sides and get

$$ady = cd - bn.$$

Voilà, we have reduced an equation with two unknowns to an equation with only one unknown, namely y. This elimination step is applied to all rows (but the last). Then, the process is repeated using as input the reduced rows.

Let us go through the whole example. The echelon form of our system is

$$\begin{pmatrix} 1 & 3 & -2 & 5 \\ 0 & 4 & -12 & 8 \\ 0 & 0 & 4 & 8 \end{pmatrix}$$

We look at the last line $0, 0, 4, 8$. We set

$$p = \frac{8}{4} = \frac{2}{1}.$$

We then call *eliminate* p on the first two lines of the matrix. Processing the first line, we compute *map* $(*1)$ *row*, which we can ignore. We then set

$$lr = 1 \times 5, al = 2 \times -2$$

and compute $lr - al$, which is 9. The complete result for the first row, hence, is $1, 3, 9$.

For the second row $0, 4, -12, 8$, *eliminate* computes

$$lr = 1 \times 8, al = 2 \times -12$$

and further computes $lr - al$, *i.e.* $8 + 24 = 32$. The complete result for the second row, hence, is $0, 4, 32$. After application of *eliminate*, xs' is thus:

$$\begin{pmatrix} 1 & 3 & 9 \\ 0 & 4 & 32 \end{pmatrix}$$

Now, in *backsub*, we repeat the process with this result. We, again, look at the last line, which now is $0, 4, 32$. We set

$$p = \frac{32}{4} = 8.$$

This goes into the result set and, as you may remember, is the result for y.

We apply *eliminate* on the remaining row, which is $1, 3, 9$. We set

$$lr = 1 \times 9 = 9, al = 8 \times 3 = 24$$

and compute $lr - al$, *i.e.* $9 - 24 = -15$. The complete result for this instance of *eliminate*, hence, is $1, -15$.

We, again, repeat the *backsub* process with this result. There is only one row left and from this line we compute p as

$$p = \frac{-15}{1} = -15,$$

which, as you may remember, is the result for x. Since *init xs* is now $[\,]$, *eliminate* will return $[\,]$ and this terminates the process with the correct result $[-15, 8, 2]$.

6.7. Binomial Coefficients are Integers

To prove that binomial coefficients are integers is quite easy. We will make our argument for coefficients of the form $1 \leq k \leq n$ in $\binom{n}{k}$. For cases outside of this range, the coefficients are defined as 1 for $k = 0$, and as 0 for $k > n$. So, there is nothing to prove in these cases.

We have the equation

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{6.52}$$

We can prove that $\binom{n}{k}$ is an integer for any n and any $k \leq \frac{n}{2}$ by induction. The induction argument holds, until we have $k > \frac{n}{2}$. At this moment, the factors in the numerator and denominator begin to intersect and the factors now common in numerator and denominator are cancelled out leading to a corresponding case in the lower half of ks, namely the case $k = n - k$. You can easily convince yourself by trying some examples that this is the reason for the symmetry in Pascal's triangle.

Since, in the proof, we have already handled the cases in the lower half of ks by induction up to $\frac{n}{2}$, there is nothing that still needs to be proven. The proof, therefore, consists in the induction argument that if $\binom{n}{k}$ is an integer, then $\binom{n}{k+1}$ is an integer too for $k \leq \frac{n}{2}$.

We first handle the trivial case $k = 1$. Here, we have $\frac{n}{1!=1}$, which, trivially, is an integer. Note that, for this case, the falling factorial, which is defined as the product of the consecutive numbers n to $n - k + 1$, is just n, since $n - 1 + 1 = n$.

For $k = 2$, we have

$$\frac{n(n-1)}{2}.$$

In the numerator we have a product of two consecutive numbers, one of which must be even. We, hence, divide the even one and the denominator by 2 and have an integer.

For $k = 3$, we have

$$\frac{n(n-1)(n-2)}{6}.$$

We now have three consecutive numbers as factors in the numerator and $3! = 6$ in the denominator. One of three consecutive numbers must be divided by 3, since there are only two numbers between any two multiples of 3. If, for instance, $n = 11$, then n and $n-1 = 10$ are not divided by 3, but $n-2 = 9$ is. So we reduce the problem to $k-1 = 2$. But note that there is a difference between the previous case $k = 2$ and the case $k - 1$ at which we are arriving now: In the previous case, we had two consecutive numbers in the numerator, but now we have three numbers that are not necessarily consecutive anymore. It may have been the middle number, $n - 1$, that we divided by 3; then, if n is odd, $n - 2$ is odd as well. However, if n and $n - 2$ are not even, then $n - 1$ must have been and then it must have been divisible by 3 and 2. In consequence, even though the numbers are not consecutive anymore, the divisibility argument still holds.

This is how the induction argument works: for any $k + 1$, we have $k + 1$ consecutive factors in the numerator and we have $(k + 1) \times k!$ in the denominator. Since there are $k + 1$ consecutive numbers in the numerator, all greater $k + 1$, one of them must be divided by $k + 1$. By dividing this number and the denominator by $k + 1$, we reduce the problem to k with k factors in the numerator and $k!$ in the denominator. Now, the factors are not consecutive anymore, but that does not affect the argument: either the number that was reduced by dividing by $k + 1$ is divided by k as well and then we have reduced the problem to $k - 2$ already, or, if it is not divided by k, then one of the other numbers must be, because we started with $k + 1$ numbers in the first place. $\square$

Let us quickly look at the next example, $k = 4$, just to illustrate the argument once again. With $k = 4$ we have the fraction

$$\frac{n(n-1)(n-2)(n-3)}{24}.$$

There are four consecutive numbers in the numerator, one of which must be divided by 4. We divide this number by 4 and the problem is reduced to the case $k = 3$. Again, the numbers are not consecutive anymore. But if the number that we reduce by dividing by 4 is divisible by 3, then we would have reduced the problem to $k = 2$ already. Otherwise, that number was just a number between two multiples of 3 and the argument does not suffer.

A concrete example makes the argument entirely clear. Consider $\binom{11}{4} = \frac{11\times10\times9\times8}{24}$. One of the numbers in the numerator must divide 4. 11 does not, 10 and 9 either, but 8 does. We divide numerator and denominator by 4 and, with that, reduce the problem to $\frac{11\times10\times9\times2}{6}$. There are 3 consecutive numbers in the numerator, one of which must be divided by 3. 11 and 10 are not divided by 3, but 9 is and, hence, we reduce the problem to $\frac{11\times 10\times 3\times 2}{2}$, which was our base case. We now have the choice to either cancel 2 in the numerator and 2 in the denominator or to divide 10 by 2 in the numerator and cancel 2 in the denominator. If we choose the latter, we divide 10 by 2 and obtain $\frac{11\times 5\times 3\times 2}{1} = 11 \times 5 \times 3 \times 2$, which is $55 \times 6 = 330$.

When calculating binomial coefficients by hand, we see that the main activity in this process is to cancel numbers in the numerator and the denominator. The question arises if we could not spare a lot of computing by using numbers that are already reduced before we start working with them. In particular, we see that the reduction of the numerator tends towards the prime factors of the binomial coefficient. We know that finding the prime factors is a very hard problem. But could there not be a shortcut to the binomial coefficient by using prime factors?

It turns out there is. The solution, however, sounds a bit surreal, since it combines facts that, on the first sight, are completely unrelated. The point is that there is a way to quickly decide for any prime number p whether it is part of the prime factorisation of a binomial coefficient and to even determine how often it occurs in its prime factorisation by counting the number of *borrows* we have to make subtracting k from n in the numeral system of base p.

To determine how often (or if at all) 2 appears in the prime factorisation of $\binom{6}{2}$, we would perform the subtraction $6 - 2$ in binary format. 6 in binary format is 110 and 2 is just 10. So, we subtract:

$$\begin{array}{cccc} & 1 & 1 & 0 \\ - & & 1 & 0 \\ = & 1 & 0 & 0 \end{array}$$

The final result 100 is 4 in decimal notation and, hence, correct. Since we have not borrowed once, 2 is not a factor of $\binom{6}{2}$. We check the next prime number 3. 6 in base 3 is 20 and 2 is just 2:

$$\begin{array}{ccc} & 2 & 0 \\ - & & 2 \end{array}$$

Now we have to borrow to compute 0 - 2, so we have:

$$\begin{array}{ccc} & 2 & 0 \\ - & \underline{1} & 2 \\ = & 1 & 1 \end{array}$$

11 base 3 is 4 in the decimal system, so the result is correct. Furthermore, we had to borrow once to compute this result. We, therefore, claim that 3 appears once in the prime factorisation of $\binom{6}{2}$.

We look at the next prime, 5. 6 in base 5 is 11 and 2 in base 5 is just 2 again. So we have:

$$\begin{array}{ccc} & 1 & 1 \\ - & & 2 \end{array}$$

Again, we have to borrow to compute 1 - 2:

$$\begin{array}{ccc} & 1 & 1 \\ - & \underline{1} & 2 \\ = & 0 & 4 \end{array}$$

Since 4 in base 5 is just decimal 4, the result, again, is correct. To reach it, we had to borrow once and we, therefore, claim that 5 appears once in the prime factorisation of $\binom{6}{2}$.

The next prime would be 7, but we do not need to go on, since 7 is greater than $n = 6$. Because we multiply n only by values less than n (namely: $n-1, n-2, \dots, n-k+1$), 7 cannot be a factor of such a number. Our final result, thus, is: $\binom{6}{2} = 3^1 \times 5^1 = 3 \times 5 = 15$. Let us check this result against our usual method to compupte the binomial coefficient: $\frac{6 \times 5}{2} = 3 \times 5 = 15$. The result is correct.

But what, on earth, has the factorisation of binomial coefficients to do with the borrows in $n - k$? The link is the following theorem:

$$\binom{n}{k} \equiv \prod_{i=0}^{r} \binom{a_i}{b_i} \pmod{p}, \tag{6.53}$$

where the as and bs are the coefficients in the representation of n and k in base p:

$$n = a_r p^r + a_{r-1} p^{r-1} + \dots + a_1 p + a_0 \tag{6.54}$$

and

$$k = b_r p^r + b_{r-1} p^{r-1} + \dots + b_1 p + b_0. \tag{6.55}$$

The theorem, hence, claims that $\binom{n}{k}$ is congruent to the product of the coefficients in the representation base p modulo that p. We are back to congruences and modular arithmetic!

The theorem is a corollary of *Lucas' theorem*, which we will now introduce as a lemma to prove the theorem above. Lucas' theorem states that

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \binom{n \bmod p}{k \bmod p} \pmod{p}, \tag{6.56}$$

which is exceptionally beautiful, since it decomposes n and k into the two parts of the Euclidian division, the quotient $\lfloor n/p \rfloor$ and the remainder $n \bmod p$. Let us rename the quotient and remainder of n and k, because we will refer to them quite often in this section: let $u = \lfloor n/p \rfloor$ and $v = n \bmod p$, such that $n = up + v$, and let $s = \lfloor k/p \rfloor$ and $t = k \bmod p$, such that $k = sp + t$. We can now rewrite the usual computation of the coefficient

$$\binom{n}{k} = \frac{n}{k} \times \frac{n-1}{k-1} \times \cdots \times \frac{n-k+1}{1} \tag{6.57}$$

as

$$\binom{n}{k} = \frac{up+v}{sp+t} \times \frac{up+v-1}{sp+t-1} \times \cdots \times \frac{up+v-k+1}{1}. \tag{6.58}$$

This formula leads to a cyclic repetition of denominators of the form

$$sp+t-1, sp+t-2, \ldots, sp+t-t.$$

We have to be careful with the denominators of the form $sp+t-t = sp$, since, modulo p, they are just zero and the corresponding fraction is thus undefined. But before we get into it, let us look at the t very first numbers, that is the fractions, before the formula reaches a multiple of p for the first time. These fractions are:

$$\frac{up+v}{sp+t} \times \frac{up+v-1}{sp+t-1} \times \cdots \times \frac{up+v-t+1}{sp+t-t+1},$$

which modulo p is

$$\frac{v}{t} \times \frac{v-1}{t-1} \times \cdots \times \frac{v-t+1}{1}.$$

This, in its turn, is just the usual way to define the binomial coefficient for v and t:

$$\binom{v}{t} = \frac{v}{t} \times \frac{v-1}{t-1} \times \cdots \times \frac{v-t+1}{1}. \tag{6.59}$$

But v and t are $n \bmod p$ and $k \bmod p$ respectively and substituting back these values for v and t in the equation leads to

$$\binom{n \bmod p}{k \bmod p} = \frac{n \bmod p}{k \bmod p} \times \frac{(n \bmod p)-1}{(k \bmod p)-1} \times \cdots \times \frac{(n \bmod p)-(k \bmod p)+1}{1} \tag{6.60}$$

and we conclude

$$\binom{n}{k} \equiv \binom{n \bmod p}{k \bmod p} X \pmod{p}, \tag{6.61}$$

where X is the rest of the product after the first t factors we are looking at right now.

Consider the example $\binom{90}{31}$ and the prime 7. For n, we get $u = 90/7 = 12$ and, since $12 \times 7 = 84$, we have $v = 90 \bmod 7 = 6$. For k, we get $s = 31/7 = 4$ and, since $4 \times 7 = 28$, we have $t = 31 \bmod 7 = 3$. The first t factors, we have looked at so far are

$$\frac{90 \times 89 \times 88}{31 \times 30 \times 29}.$$

We take all factors in numerator and denominator modulo 7:

$$\frac{6 \times 5 \times 4}{3 \times 2 \times 1},$$

which, as you can see, is just $\binom{6}{3} = \binom{90 \bmod 7}{31 \bmod 7}$.

Now we will look at the ominous X. Since X is the product with the first $k \bmod p$ factors cut off and the number of factors in the entire product is k, the number of the remaining factors is a multiple of p. These factors fall into $\frac{k}{p}$ groups each of which contains in the numerator and the denominator one multiple of p and $p-1$ remainders of p. Let us look at the denominators of such a group:

$$\frac{\dots}{sp} \times \frac{\dots}{sp+1} \times \dots \times \frac{\dots}{sp+p-1}.$$

Since the whole is a product, these values are multiplied with each other and, as we know from Wilson's theorem, the factorial of $p-1$ is $(p-1)! \equiv p-1 \pmod{p}$. We certainly have the same remainders in the numerators modulo p, which, again according to Wilson, are $p-1$ modulo p and, therefore, cancel out. We are then left with the factors that are multiples of p.

Continuing the example, the first of such groups would be

$$\frac{87 \times 86 \times 85 \times 84 \times 83 \times 82 \times 81}{28 \times 27 \times 26 \times 25 \times 24 \times 23 \times 22}$$

Here, 84 in the numerator and 28 in the denominator are multiples of 7. All other numbers are remainders. In the denominator, we have a complete group of remainders $22 \dots 27$, which are $1 \dots 6$ modulo 7. These multiplied with each other are, according to Wilson's theorem, congruent to 6 modulo 7. In the numerator, we do not see the whole group at once. Instead, we see two different parts of two groups separated by 84, the multiple of 7, in the middle: $85, 86, 87$, which are congruent to 1, 2 and 3 modulo 7, and $81, 82, 83$, which are congruent to 4, 5 and 6 modulo 7. Multiplying these remainders is, again according to Wilson's theorem, congruent to 6 modulo 7. So, we cancel all these values in numerator and denominator and keep only

$$\frac{84}{28}.$$

As already observed, we have $\lfloor k/p \rfloor = \lfloor 31/7 \rfloor = 4$ of such groups. We are therefore left with 4 fractions with a multiple of 7 in the numerator and the denominator, namely the factors:

$$\frac{84}{28} \times \frac{77}{21} \times \frac{70}{14} \times \frac{63}{7}.$$

When we divide numerator and denominator by 7, we get

$$\frac{12}{4} \times \frac{11}{3} \times \frac{10}{2} \times \frac{9}{1}.$$

and see by this simple trick of black magic that the result is

$$\binom{12}{4} = \binom{\lfloor 90/7 \rfloor}{\lfloor 31/7 \rfloor} = \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}.$$

Unfortunately, there are very few scholars left who would accept magic as proof and so we must continue with the abstract reasoning. Note again that we have taken the first $t = k \bmod p$ terms out in the previous step. The denominators we are left with, when we arrive at fractions with multiples of p in the numerator and denominator, are therefore $k - (k \bmod p), k - (k \bmod p) - p, k - (k \bmod p) - 2p, \dots, 1$. In the example above, 28 corresponds to $k - (k \bmod p)$: $31 - 3 = 28$, 21 corresponds to $k - (k \bmod p) - p$ and so on.

The numerators are not so clean, but very similar: $n - (k \bmod p) - x, n - (k \bmod p) - x - p, \dots$ The x in this formula results from the fact that $n - (k \bmod p)$ does not necessarily result in a multiple of p. For instance, $90 - 3 = 87$ is not a multiple of p. x in this case is 3, since $90 - 3 - 3 = 84$, which is a multiple of p. In fact, we can determine the value of x more specifically as $(n \bmod p) - (k \bmod p)$, which is $6 - 3 = 3$, but we do not need to make use of this fact. It is sufficient to realise that each value must be divisible by p and, hence, $(k \bmod p) + x < p$. When we now divide by p, we get for each factor

$$\frac{\lfloor (n - (k \bmod p) - x_i - a_i p)/p \rfloor}{\lfloor (k - (k \bmod p) - b_i p)/p \rfloor},$$

where the as and bs run from 0 to the number of groups we have minus 1, *i.e.* $\lfloor k/p \rfloor - 1$.

Since the second term of the differences in numerator and denominator are remainders of p that, together with n and, respectively, k, are multiples of p, this is just the same as saying

$$\frac{\lfloor (n - a_i p)/p \rfloor}{\lfloor (k - b_i p)/p \rfloor},$$

which of course is

$$\frac{\lfloor n/p \rfloor - a_i}{\lfloor k/p \rfloor - b_i}.$$

Since the as and bs run from 0 to the number of the last group, we get this way the product

$$\frac{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \times \frac{\lfloor n/p \rfloor - 1}{\lfloor k/p \rfloor - 1} \times \frac{\lfloor n/p \rfloor - 2}{\lfloor k/p \rfloor - 2} \times \dots \times \frac{\lfloor n/p \rfloor - \lfloor k/p \rfloor + 1}{\lfloor k/p \rfloor - \lfloor k/p \rfloor + 1},$$

which we immediately recognise as the computation for

$$\binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}.$$

You, hopefully, remember that this is the X, we left over in equation 6.61. Substituting for X we derive the intended result:

$$\binom{n}{k} \equiv \binom{n \bmod p}{k \bmod p} \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor} \pmod{p} \tag{6.62}$$

and this completes the proof. □

But we have to add an important remark. Binomial coefficients with $k > n$ are defined to be zero. The equation, thus, tells us that the prime p divides the coefficient if $k \bmod p > n \bmod p$. For instance $\binom{8}{3}$, which is $\frac{8 \times 7 \times 6}{6} = 8 \times 7 = 56$, is divided by 7, since 7 appears as a factor in the numerator and, indeed: $\binom{8 \bmod 7}{3 \bmod 7} = \binom{1}{3}$. This would also work with $\binom{9}{3}$, which is $\frac{9 \times 8 \times 7}{6} = 3 \times 4 \times 7 = 84$, where 7, again, appears as a factor in the numerator and $\binom{9 \bmod 7}{3 \bmod 7} = \binom{2}{3}$. It does not work with $\binom{9}{2}$, which is $\frac{9 \times 8}{2} = 9 \times 4 = 36$, since $\binom{9 \bmod 7}{2 \bmod 7} = \binom{2}{2} = 1$ and, indeed: $36 \bmod 7 = 1$. Let us memorise this result: a prime p divides a binomial coefficient $\binom{n}{k}$, if $k \bmod p > n \bmod p$.

We, finally, come to the corollary, which we wanted to prove in the first place. We need to prove that

$$\binom{n}{k} \equiv \prod_{i=0}^{r} \binom{a_i}{b_i} \pmod{p}, \tag{6.63}$$

where the as and bs are the coefficients in the representation of n and k base p. We now calculate u, v, s and t, as we have done before, as $u = \lfloor n/p \rfloor$, $v = n \bmod p$, which is just the last coefficient in the p-base representation of n a_0, $s = \lfloor k/p \rfloor$ and $t = k \bmod p$, which is just the last coefficient b_0.

The p-base representations of n and k are $n = a_r p^r + \dots + a_1 p + a_0$ and $k = b_r p^r + \dots + b_1 p + b_0$. If we divide those by p, we get $u = a_r p^{r-1} + \dots + a_1$ and $s = b_r p^{r-1} + \dots + b_1$ with a_0 and b_0 as remainders.

From Lucas' theorem we conclude that

$$\binom{n}{k} \equiv \binom{u}{v} \binom{a_0}{b_0} \pmod{p}. \tag{6.64}$$

Now we just repeat the process for u and v:

$$\binom{n}{k} \equiv \binom{\lfloor u/p \rfloor}{\lfloor v/p \rfloor} \binom{a_1}{b_1} \binom{a_0}{b_0} \pmod{p} \tag{6.65}$$

and continue until we have

$$\binom{n}{k} \equiv \binom{a_r}{b_r} \dots \binom{a_1}{b_1} \binom{a_0}{b_0} \pmod{p}, \tag{6.66}$$

which then concludes the proof. □

We see immediately that p divides $\binom{n}{k}$, when at least one digit of k in the p-base representation is greater than the corresponding digit of n, because, in this case, the corresponding binomial coefficient is zero and, in consequence, the whole product modulo p becomes zero.

That a digit of k is greater than the corresponding digit of n implies that, on subtracting k from n, we have to borrow from the next place. Therefore, if we have to borrow during subtraction, then p divides $\binom{n}{k}$ and, thus, is a prime factor of that number.

So, we divide $\binom{n}{k}$ by p leading to the product in equation 6.66 with one pair of digits removed and search again for a pair of digits where $k > n$. If we find one, the number $\binom{n}{k}$ is divided twice by p, so p appears twice in the factorisation of that number. We again divide by p and repeat the process until we do not find a pair $k > n$ anymore. Then we know how often p appears in the prime factorisation of $\binom{n}{k}$. If we do this for all primes $\leq n$, we learn the complete prime factorisation of $\binom{n}{k}$.

We now will implement this logic in Haskell. We start with the notion of borrows:

```
borrows :: Natural → Natural → Natural → Natural → Natural → Natural
borrows _ 0 _ _ _ = 0
borrows p u v s t | v < t = 1 + borrows p (u `div` p) (u `rem` p)
                                          (s `div` p) ((s `rem` p) + 1)
                  | otherwise = borrows p (u `div` p) (u `rem` p)
                                          (s `div` p) (s `rem` p)
```

The function *borrows* takes five arguments all of our old type *Natural*. The first argument is the prime; the next four arguments are u, v, s and t, that is $u = \lfloor n/p \rfloor$, $v = n \bmod p$, $s = \lfloor k/p \rfloor$ and $t = k \bmod p$.

If $u = 0$, we are through and no more borrows are to be found. Otherwise, if $v < t$, we have to borrow. The borrow is actually seen in the recursive call of borrows, where we increment $s \bmod p$ by 1. We also add 1 to the overall result. Otherwise, we call borrows with the quotients and remainders of u and s. The recursion implements the logic we see in equation 6.66: we reduce the product factor by factor by dividing by p; on each step, we check if a borrow occurs and continue with the next step.

This is the heart of our algorithm. However, we can improve on this. There are cases we can decide immediately without going through the whole process. Consider a prime $p \leq n - k$. Since the numerator in the computation of the binomial coefficient runs from $n \dots (n - k + 1)$, this prime will not appear directly in the numerator. It could of course still be a factor of one of the numbers $n, n-1, \dots, n-k+1$ and, as such, be a factor of the resulting number. But then it must be less than or at most equal to the half one of those numbers. Otherwise, there would be no prime number by which we could multiply p to obtain one of those number. In consequence, when $p \leq n - k$ and $p > n/2$, p cannot divide $\binom{n}{k}$.

On the other hand, if $p > n - k$, then it appears in the numerator; if also $p > k$, then it will not appear in the denominator. In consequence, if $p > k$ and $p > n - k$, then it will not be cancelled out and is therefore prime factor of the binomial coefficient. Furthermore, it can appear only once in the numerator. There are only k consecutive numbers in the numerator and $p > k$. If we assume there is a factor ap with $a \geq 1$, then we will reach n in one direction and $n - k + 1$ in the other direction, before we reach either $(a + 1)p$ or $(a - 1)p$. Therefore, $a = 1$ or, in other words, p appears only once in the prime factorisation.

Finally, if $n \bmod p < k \bmod p$, we know for sure that p divides the number at least once. If $p^2 > n$, then we know that p divides $\binom{n}{k}$ exactly once.

We will implement these one-step decisions as filters in a function that calls *borrows*:

```
powOfp :: Natural → Natural → Natural → Natural
powOfp n k p | p ⩽ n − k ∧ p > n ‘div‘ 2          = 0
             | p > k ∧ p > n − k                   = 1
             | p ∗ p > n ∧ n ‘rem‘ p < k ‘rem‘ p = 1
             | otherwise = borrows p (n ‘div‘ p) (n ‘rem‘ p)
                                     (k ‘div‘ p) (k ‘rem‘ p)
```

Now we implement a new variant of *choose* making use of borrows:

```
choose3 :: Natural → Natural → Natural
choose3 n k  = product (map f ps)
  where ps   = takeWhile (⩽ n) allprimes
        f p  = p ↑ (powOfp n k p)
```

The implementation is quite simple and there is certainly room for optimisation. It just maps $p^{(\text{powOfp}\ n\ k\ p)}$ on all primes up to n and builds the product of the result. A possible improvement is to use *fold* instead of *map* and not to add primes to the result for which *powOfp* yields 0. That would reduce the size of the resulting list of primes drastically. With *map*, there are a lot of 1s, in fact, most of the elements are 1 in most cases.

But let us investigate the running time of *choose3* compared to that of $\frac{n^k}{k!}$ in more general terms. The running time of the fraction is a function of k, such as $2k - 1$, since we have $k - 1$ multiplications in numerator and denominator and one division, as already discussed in a previous chapter. Since we are not too much interested in the details of the operations, *i.e.* the cost of single multiplications and divisions and the cost of *borrows* for one prime, we will use the *big-O* notation, for instance: $\mathcal{O}(2k - 1)$. This notation tells us that there is a function of the form $2k - 1$, which is a limit for our function, the running time of the fraction. In other words, for huge input values, the function within the $\mathcal{O}$ is equal to or greater than our function.

The running time of *choose3*, by contrast, is a function of $\Pi(n)$, that is the number of primes up to n, approxmimately $n/\ln n$. In big-O notation, we state that the running time of *choose3* is $\mathcal{O}(n/\ln n)$.

For large ns and small ks, *e.g.* $\binom{1000000}{2}$, the fraction appears to be much better. Of course, the multiplications in the numerator are heavy, since the complexity of multiplication grows with the number of digits of the factors, but there are still only few such multiplications. The multiplications in the denominator, in compensation, are trivial with small ks.

The complexity of *choose3* for this specific number is $1000000/\ln 1000000 \approx 72382$ and, as such, of course much worse. With larger ks, however, the picture changes. Even though we can reduce the complexity of the fraction for cases where $k > n/2$ to cases with $k < n/2$, as discussed before, there is sufficient room for ks around $n/2$ that makes *choose3* much more efficient than the fraction. In general, we can say that *choose3* is more efficient, whenever $\frac{n}{2\ln n} + 1 < k < n - \frac{n}{2\ln n} - 1$. For the specific example of

$n = 1000000$, *choose3* is more efficient for $36192 < k < 963807$.

The fact underlying the algorithm, the relation between the number of occurrences of a prime p in the factorisation of a binomial coefficient and the number of borrows in the subtraction of n and k in base-p, was already known in the 19^{th} century, but was apparently forgotten during the 20^{th} century. It was definitely mentioned by German mathematician Ernst Kummer (1810 – 1893), but described as an algorithm apparently only in 1987 in a short, but nice computer science paper by French mathematician Pascal Goetgheluck who rediscovered the relation by computer analysis.

Lucas' theorem is named for Édouard Lucas (1842 – 1891), a French mathematician who worked mainly in number theory, but is also known for problems and solutions in recreational math. The *Tower of Hanoi* is of his invention. Lucas died of septicemia in consequence of a household accident where he was cut by a piece of broken crockery.

6.8. Euler's Totient Function

The constructor function *ratio* of our *Ratio* data type reduces fractions to their canonical form by dividing numerator and denominator by their *gcd*. If the *gcd* is just 1, numerator and denominator do not change. If the *gcd* is not 1, however, the numbers actually do change. For instance, the fraction $\frac{1}{6}$ is already in canonical form. The fraction $\frac{3}{6}$, however, is not, since $gcd(6,3)$ is 3 and so we reduce: $\frac{3/3=1}{6/3=2} = \frac{1}{2}$.

This leads to the observation that not all fractions possible with one denominator manifest with the *Ratio* number type. For the denominator 6, for example, we have only $\frac{1}{6}$ and $\frac{5}{6}$. For other numerators, we have $\frac{2}{6} = \frac{1}{3}$, $\frac{3}{6} = \frac{1}{2}$ and $\frac{4}{6} = \frac{2}{3}$. We could write a function that shows all proper fractions for one denominator, *e.g.*

```
fracs1 :: Natural → [Ratio]
fracs1 n = [x % n | x ← [1 .. n − 1]]
```

fracs1 6, for instance, gives:

$[Q\ 1\ 6, Q\ 1\ 3, Q\ 1\ 2, Q\ 2\ 3, Q\ 5\ 6]$,

which is easier to read in mathematical notation:

$$\frac{1}{6}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{5}{6}.$$

How many proper fractions are there for a specific denominator? In the example above, there are only two proper fractions with the denominator 6. We devise a function to filter out the fractions that actually preserve the denominator 6:

```
fracs2 :: Natural → [Ratio]
fracs2 n = filter (λ(Q _ d) → d ≡ n) (fracs1 n)
```

This function, again applied to 6 would give (in mathematical notation):

$$\frac{1}{6}, \frac{5}{6}.$$

Applied to 12, it would give:

$$\frac{1}{12}, \frac{5}{12}, \frac{7}{12}, \frac{11}{12}.$$

It is easy to see that the numerators of the fractions with a given denominator n, correspond to the group of numbers $g < n$ coprime to n. We, hence, could also create a function that just finds the coprimes from the range $0 \dots n-1$:

```
coprimes :: Natural → [Natural]
coprimes n = [x | x ← [0 .. n − 1], gcd n x ≡ 1]
```

With little surprise, we see that *coprimes* 12 gives

$$1, 5, 7, 11.$$

Mathematicians like to quantify things and so it is no wonder that there is a well known function, Euler's *totient* function, often denoted $\varphi(n)$, that actually counts the coprimes. This is easily implemented:

```
tot :: Natural → [Natural]
tot = fromIntegral ∘ length ∘ coprimes
```

Let us have a look at the results of the totient function for the first 20 or so numbers:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	...
1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	...

We first see that the totient of a prime p is $p-1$: $\varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(5) = 4$, $\varphi(7) = 6$, $\varphi(11) = 10, \dots$ Of course, that is the group of remainders of that prime – the previous chapter was dedicated almost entirely to that group!

The totients of composites are different. They slowly increase, but many numbers have the same totient. Since the difference between primes and composites lies in the fact that composites have divisors different from 1 and themselves, it is only natural to suspect that there is a relation between the totients of composites and their divisors. For instance, 2 has one divisor: 1. The totien of 1 is $\varphi(1) = 1$ and that of 2, $\varphi(2)$ is 1 as well. It may just be one of those peculiarities we see with small numbers, but what we see is the curious relation $\varphi(1) + \varphi(2) = 2$. We could formulate the hypothesis that the sum of the totients of the divisors of a number n is that number n. $n = \varphi(1)+\varphi(2)+\dots+\varphi(n)$ or, more elegantly:

$$n = \sum_{d|n} \varphi(d). \tag{6.67}$$

Let us check this hypothesis. From 2, we go on to 3, but 3 is prime and has only the divisors 1 and 3 and, trivially, $\varphi(1) + \varphi(3) = 3$, since, for any prime p: $\varphi(p) = p - 1$. We go on to 4. The divisors of 4 are 1,2 and 4: $\varphi(1) + \varphi(2) + \varphi(4) = 1 + 1 + 2 = 4$. The next number, 5, again is prime and we go on to 6, which has the divisors 1, 2, 3, 6: $\varphi(1)+\varphi(2)+\varphi(3)+\varphi(6) = 1+1+2+2 = 6$. Until here, the equation is confirmed. Let us jump a bit forward: 12 has the divisors 1, 2, 3, 4, 6, 12: $\varphi(1) + \varphi(2) + \varphi(3) + \varphi(4) + \varphi(6) + \varphi(12) = 1 + 1 + 2 + 2 + 2 + 4 = 12$. The equation appears to be true. But can we prove it?

In fact, it follows from a number of fundamental, but quite simple theorems that one would probably tend to take for granted on first encounter. One of these theorems is related to cardinality of set union. The theorem states that

$$|S1 \cup S2| = |S1| + |S2| - |S1 \cap S2| \tag{6.68}$$

that is: the cardinality of the union of two sets equals the sum of the cardinalities of the two sets minus the cardinality of the intersection of the two sets.

Proof: The intersection of two sets $S1$ and $S2$ contains all elements that are both in $S1$ and $S2$. The union of two sets contains all elements of $S1$ and $S2$. But those elements that are in both sets will appear only once in the union, since this is the definition of the very notion of *set*. We can therefore first build a collection of all elements in the sets including the duplicates and then, in a second step, remove the duplicates. The elements that we remove, however, are exactly those that are also elements of the intersection. The number of elements in the union, hence, is exactly the sum of the numbers of elements of the individual sets minus the number of duplicates. □

There are two corollaries that immediately follow from this theorem. First, for two disjoint sets $S1$ and $S2$, *i.e.* sets for which $S1 \cap S2 = \varnothing$, the equation above simplifies to:

$$|S1 \cup S2| = |S1| + |S2|. \tag{6.69}$$

This is trivially true, since $|\varnothing| = 0$.

Second, for sets that are pairwise disjoint (but only for those!), we can derive the general case:

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{i=1}^{n} |S_i|, \tag{6.70}$$

where $\bigcup$ is the union operator for n sets, where n is not necessarily 2. It, hence, does for $\cup$ what $\sum$ does for $+$. Systems of such operators that are applied to an arbitrary number of operands are called σ-algebras. But, for the time being, that is just a fancy word.

The next fundamental theorem states that the sum of the divisors of a number n equals the sum of the fractions $\frac{n}{d_i}$, where $d_i = d_1, d_2, \dots, d_r$ are the divisors of n. More formally, the theorem states:

$$\sum_{d|n} d = \sum_{d|n} \frac{n}{d}. \tag{6.71}$$

The point, here, is to see that if d is a divisor of n, then $\frac{n}{d}$ is a divisor too. That d is a divisor means exactly that: n divided by d results in another integer m, such that $d = \frac{n}{m}$ and $dm = n$. Since the set of divisors of n contains all divisors of n and the set of quotients $\frac{n}{d}$ contains quotients with all divisors, the two sets are equal. The only aspect that changes, when we see these sets as sequences of numbers, is the order. Since order has no influence on the result of the sum, the two sums are equal. □

For the example $n = 12$, the divisors are $1, 2, 3, 4, 6, 12$. The quotients generated by dividing n by the divisors are $12, 6, 4, 3, 2, 1$. The sum of the first sequence is $1 + 2 + 3 + 4 + 6 + 12 = 28$. The sum of the second sequence is $12 + 6 + 4 + 3 + 2 + 1 = 28$.

It remains to note that, when we have a sum of functions, then still $\sum_{d|n} f(\frac{n}{d}) = \sum_{d|n} f(d)$, since the values to which the function is applied are still the same in both sets.

Equipped with these simple tools, we return to the sum of the totients of the divisors. We start by defining a set S_d that contains all numbers, whose *gcd* with n is d:

$$S_d = \{m \in \mathbb{N} : 1 \leq m \leq n, \gcd(n, m) = d\}. \tag{6.72}$$

In Haskell this would be:

```
s :: Natural → Natural → [Natural]
s n d = [m | m ← [1 .. n], gcd n m ≡ d]
```

When we map this function with $n = 12$ on the divisors of 12, *map s* 12 $[1, 2, 3, 4, 6, 12]$, we get:

```
[1,5,7,11]
[2,10]
[3,9]
[4,8]
[6]
[12].
```

We see six pairwise disjoint sets whose union equals the numbers $1 \dots 12$. The first set contains the coprimes of 12, since we ask for m, such that $\gcd(12, m) = 1$. The next set contains the numbers, such that $\gcd(12, m) = 2$, the next, the numbers, such that $\gcd(12, m) = 3$ and so on. In other words, these lists together contain all numbers $1 \dots 12$ partitioned according to their greatest common divisor with $n = 12$. Note that the lists together necessarily contain all the numbers in the range $1 \dots n$, since, either a number does not have common divisors with n, then it is in the first set for $\gcd(n, m) = 1$, or it has a common divisor with n. Then it is in one of the other sets. This is just what the set S_d mapped on the divisors of n is about.

The sets are also necessarily disjoint from each other, since no number m would, on one occasion, have a *gcd* d_1 with n and, on another, a distinct *gcd* d_2 with the same n. It either shares d_1 as greatest common divisor with n or divisor d_2. It, hence, is either in set S_{d_1} or in set S_{d_2}.

But there is more. The set for divisor 2 contains 2 and 10. These numbers divided by 2 give 1 and 5. $\frac{12}{2}$ is 6 and 1 and 5 are the coprimes of 6. The set for divisor 3 contains 3 and 9; these numbers divided by 3 are 1 and 3. $\frac{12}{3}$ is 4 and 1 and 3 are the coprimes of 4 and so on. In other words, the sets that we see above contain numbers m that, divided by the corresponding divisor d, $\frac{m}{d}$, are the coprimes of $\frac{n}{d}$. This results from the fact that these numbers and the divisor are related to each other by the gcd. When we have two numbers m and n and we compute their gcd: $d = \gcd(n, m)$, then $\frac{n}{d}$ is coprime to $\frac{m}{d}$, since we divide them by the biggest number that divides both. Therefore, all numbers in the set S_d are necessarily coprimes of $\frac{n}{d}$.

Can there be a coprime of $\frac{n}{d}$ (less than $\frac{n}{d}$) that is not in the set S_d? We created the list of coprimes by first computing m, such that $\gcd(n, m) = d$, and then $c = \frac{m}{d}$. Now, let us assume that there is a coprime c that escapes this filter. In other words, there is another number $k \neq d$, such that $\gcd(n, m) = k$ and $c = \frac{m}{k}$. To be a coprime of interest, we must have $\frac{m}{k} < \frac{n}{d}$. Since $\frac{dn}{d} = n$, we must have $\frac{dm}{k} < n$. This number must therefore appear in one of the S_{d_i}. We can ask: in which one? The answer is $\gcd(n, \frac{dm}{k})$. There are two candidates: d and $\frac{m}{k}$. But $\frac{m}{k}$ cannot be a divisor of n, since k is the greatest divisor m and n have in common. They do not share any other divisor, not even $\frac{m}{k}$. Therefore, d must be the greatest common divisor of n and $\frac{dm}{k}$. But then this number appears in S_d and $\frac{m}{k}$ does not escape our filter.

It follows that each of the sets S_d contains exactly those numbers that divided by d are the coprimes of $\frac{n}{d}$. The size of each of these sets is thus the totient number of $\frac{n}{d}$:

$$|S_d| = \varphi\left(\frac{n}{d}\right). \tag{6.73}$$

To complete the proof, we now have to extend the relation between one of those sets and the totient of one $\frac{n}{d}$ to that between the union of all the S_{d_i} and the sum of the totient numbers of the divisors. From cardinality of disjoint sets (equation 6.70) we know that the cardinality of the union of disjoint sets is the sum of the cardinality of each of the sets, so we have:

$$\left|\bigcup_{d|n} S_d\right| = \sum_{d|n} \varphi\left(\frac{n}{d}\right). \tag{6.74}$$

From sum of divisors (equation 6.71) we know even further that the sum of $\frac{n}{d}$ equals the sum of d, therefore:

$$\left|\bigcup_{d|n} S_d\right| = \sum_{d|n} \varphi(d). \tag{6.75}$$

We have seen that the union of the S_{d_i} for a given n contains all numbers in the range $1 \dots n$:

$$\bigcup_{d|n} S_d = \{1 \dots n\}. \tag{6.76}$$

Since the set $\{1 \dots n\}$ contains n numbers, we can conclude that

$$\left|\bigcup_{d|n} S_d\right| = n, \tag{6.77}$$

from which, together with equation 6.75, we then can conclude that

$$\sum_{d|n} \varphi(d) = n. \quad \Box \tag{6.78}$$

We could define a recursive function very similar to Pascal's rule that exploits this relation. We first define a function to get the divisors

```
divs :: Natural → [Natural]
divs n = [d | d ← [1 .. n], rem n d ≡ 0]
```

Then we add up the totients of these numbers (leaving n out, because that is the one we want to compute) and subtract the result from n and, this way, obtain the totient number of n:

```
divsum :: Natural → [Natural]
divsum 1 = 1
divsum 2 = 1
divsum n = n − sum [divsum d | d ← divs n, d < n]
```

Another property of the totient function is multiplicity of totients of coprimes, that is

$$\varphi(a) \times \varphi(b) = \varphi(ab), \text{if} \gcd(a, b) = 1. \tag{6.79}$$

For instance, the coprimes of 3 are 1 and 2; those of 5 are 1, 2, 3 and 4. $\varphi(3)$, hence, is 2 and $\varphi(5)$ is 4. $\varphi(3 \times 5 = 15)$ is 8, which also is 2×4. Indeed, the coprimes of 15 are 1, 2, 4, 7, 8, 11, 13 and 14. An example of two coprimes that are not both primes is 5 and 8. $\varphi(5) = 4$ and $\varphi(8) = 4$. $\varphi(5 \times 8 = 40) = 16$, which also is 4×4.

This property might look surprising at the first sight, but is becomes almost trivial in the light of the Chinese Remainder theorem. For two coprimes a and b and their sets of coprimes A and B, we can, for any $a_i \in A$ and $b_j \in B$ create congruence systems of the form

$$\begin{aligned} x &\equiv a_i \pmod{a} \\ x &\equiv b_j \pmod{b} \end{aligned}$$

The Chinese Remainder theorem guarantees that, for every case, their is a solution that is unique modulo ab, *i.e.* there are no two different systems with the same solution and there is no system without a solution. Since the solutions are unique modulo ab, there must be exactly one number in the group of coprimes of ab for any combination of a_i and b_j. Since there are $|A| \times |B|$ combinations of all elements of A and B, $\varphi(ab)$ must be $\varphi(a) \times \varphi(b)$.

To illustrate that, we can create all the combinations of as and bs and then apply the Chinese remainder on all of them. First we create all combinations of as and bs:

```
consys :: Natural → Natural → [[Natural]]
consys a b = concatMap mm (coprimes b)
  where mm y = [[x, y] | x ← coprimes a]
```

The result for *consys* 5 8, for instance, is:

```
[1,1],[2,1],[3,1],[4,1],
[1,3],[2,3],[3,3],[4,3],
[1,5],[2,5],[3,5],[4,5],
[1,7],[2,7],[3,7],[4,7]
```

Now we map *chinese* on this:

```
china :: Natural → Natural → [Natural]
china a b = map (esenihc [a, b]) (consys a b)
  where esenihc = flip chinese
```

Note that, to map *chinese* on *consys*, we have to flip it. *chinese* expects first the congruences and then the moduli, but we need it the other way round.

This is the result for *china* 5 8:

$1, 11, 21, 31, 17, 27, 37, 7, 33, 3, 13, 23, 9, 19, 29, 39$

and this is the result for *coprimes* 40:

$1, 3, 7, 9, 11, 13, 17, 19, 21, 23, 27, 29, 31, 33, 37, 39.$

When you sort the result of *china* 5 8, you will see that the results are the same.

We are now approaching the climax of this section. There is a little fact we need, before we can go right to it, which may appear a tiny curiosity. This curiosity is yet another property of the totient function concerning the totient of powers of prime numbers:

$$\varphi(p^k) = p^k - p^{k-1}. \tag{6.80}$$

That is, if p is prime, then the totient of p^k equals the difference of this number p^k and the previous power of that prime p^{k-1}. An example is 27, which is 3^3. We compute $27 - 3^2 = 27 - 9 = 18$, which is indeed $\varphi(27)$.

The proof is quite simple. Since p is prime, its powers have only one prime factor, namely p. When we say *prime power*, we mean exactly this: a number whose factorisation consists of one prime raised to some $k \geq 1$: p^k. Therefore, the only numbers that share divisors with p^k are multiples of p less than or equal to p^k: p, $2p$, $3p$, $\dots$, p^k. The 9 numbers that share divisors with 27 are:

$$3, 6, 9, 12, 15, 18, 21, 24, 27.$$

How many multiples of p less than or equal to p^k are there? There are p^k numbers in the range $1 \dots p^k$, every p^{th} number of which is a multiple of p. There, hence, are $\frac{p^k}{p} = p^{k-1}$ numbers that divide p^k. The number of coprimes in this range is therefore p^k minus that number and, thus, $p^k - p^{k-1}$. □

We can play around a bit with this formula. The most obvious we can do is to factor p^{k-1} out to get $p^{k-1}(p-1)$. So, we could compute $\varphi(27) = 9 \times 2 = 18$. Even more important for the following, however, is the formula at which we arrive by factoring p^k out. We then get

$$\varphi\left(p^k\right) = p^k\left(1 - \frac{1}{p}\right) \tag{6.81}$$

This formula leads directly to a closed form for the totient function, namely *Euler's product formula*. Any number can be represented as a product of prime powers: $n = p_1^{k_1}p_2^{k_2}\dots$ Since the ps in this formula are all prime, the resulting prime powers are for sure coprime to each other. That means that the multiplicity property of the totient function applies, *i.e.*

$$\varphi(n) = \varphi\left(p_1^{k_1}\right)\varphi\left(p_2^{k_2}\right)\dots.$$

We now can substitute the totient computations of the prime powers on the right-hand side by the formula in equation 6.81 resulting in

$$\varphi(n) = p_1^{k_1}\left(1 - \frac{1}{p_1}\right)p_2^{k_2}\left(1 - \frac{1}{p_2}\right)\dots$$

We regroup the formula a bit to get:

$$\varphi(n) = p_1^{k_1}p_2^{k_2}\dots(1 - \frac{1}{p_1})\left(1 - \frac{1}{p_2}\right)\dots$$

and see that we have all the prime factors of n and then the differences. The prime factors multiplied out result in n, so we can simplify and obtain Euler's product formula:

$$\varphi(n) = n\prod_{p|n}\left(1 - \frac{1}{p}\right). \tag{6.82}$$

Consider again the example $n = 12$. The formula claims that

$$\varphi(12) = 12 \times \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right),$$

since the prime factors of 12 are 2 and 3. Let us see: $1 - \frac{1}{2}$ is just $\frac{1}{2}$, $1 - \frac{1}{3}$ is $\frac{2}{3}$. We therefore get $12 \times \frac{1}{2} \times \frac{2}{3}$. $12 \times \frac{1}{2} = 6$ and $6 \times \frac{2}{3} = \frac{12}{3} = 4$, which, indeed, is $\varphi(12)$.

We can use this formula to implement a variant of *tot* in Haskell:

```
ptot :: Natural → Natural
ptot n = let r = ratio n 1
         in  case r * product [ratio (p − 1) p | p ← nub (trialfact n)] of
               Q t 1 → t
               _     → error "Totient not an integer"
```

Note that we use the *Ratio* number type. To convert it back to *Natural*, we check if the denominator is 1, *i.e.* if the resulting fraction is indeed an integer. If so, we return the numerator. Otherwise, we create an error, which, of course, should not occur if Euler was right.

But what is so special about this function? Well, it leads us quickly to an old friend. Consider a squarefree number n, that is a number where the exponents in the prime factorisation are all 1: $n = p_1p_2 \dots p_r$. We perform some simple transformations on 6.82. First, we transform the factors $1 - \frac{1}{p}$ to $\frac{p}{p} - \frac{1}{p} = \frac{p-1}{p}$. Then we have:

$$\varphi(n) = n\frac{p-1}{p}\frac{q-1}{q}\dots \tag{6.83}$$

As you may have noticed, this is the form in which we implemented Euler's formula in *ptot*. Now we multiply this out:

$$\varphi(n) = \frac{n(p-1)(q-1)\dots}{pq\dots} \tag{6.84}$$

Since n is squarefree, the denominator, the product of the prime factors, equals n. n and the entire denominator, hence, cancel out. Now we have:

$$\varphi(n) = (p-1)(q-1)\dots \tag{6.85}$$

That is, the totient number of a squarefree number is the product of the prime factors, each one reduced by one, a result that follows from multiplicity of totients of numbers that are coprime to each other.

The formula is quite interesting. It is very close to the formula we used for the RSA algorithm to find a number t for which

$$a^t \equiv 1 \pmod{n}.$$

It will probably not come as a surprise that there is indeed *Euler's theorem*, which states that, for a and n coprime to each other:

$$a^{\varphi(n)} \equiv 1 \pmod{n}, \tag{6.86}$$

which, as you will at once recognise, is a generalisation of Fermat's little theorem. Fermat's theorem expresses the same congruence for a prime number. Since, as we know, the totient number of a prime number p is $p - 1$, Fermat's theorem can be reduced to Euler's theorem. In fact, Euler's theorem is nothing new at all. We already know from the previous chapter that the powers of a up to a^k, such that $a^k \equiv 1 \pmod{n}$, establish a group or subgroup of the numbers coprime to n. Since $\varphi(n)$ is the size of the group of coprimes of n, any subgroup of n has either size $\varphi(n)$ or a size that divides $\varphi(n)$. But for sure, for any coprime a the group is at most $\varphi(n)$, since there are only $\varphi(n)$ elements in the group.

For the example 12, the groups are all trivial. The coprimes of 12 are 1, 5, 7 and 11. 1 establishes the trivial group $\{1\}$. 5 is in the likewise trivial group $\{1, 5\}$, since $5^2 = 25$ and, hence, $5^2 \equiv 1 \pmod{12}$. Since $7^2 = 49 \equiv 1 \pmod{12}$, the group of 7 is also trivial. Finally, since $11^2 = 121 \equiv 1 \pmod{12}$, 11 is in a trivial group as well.

A more interesting case is 14. Let us look at the groups of 14 using *generate*, which we defined in the previous chapter, like *map* (*generate* 14) (*coprimes* 14):

```
[1]
[3,9,13,11,5,1]
[5,11,13,9,3,1]
[9,11,1]
[11,9,1]
[13,1]
```

We see four different groups. The trivial groups 1 and $n - 1$ and the non-trivial groups $\{1, 3, 5, 9, 11, 13\}$, identical to the coprimes of 14, and $\{1, 9, 11\}$, a subgroup with three elements.

From these examples it should be clear that not for all coprimes a $\varphi(n)$ is the first number for which the congruence in Euler's theorem is established. In fact, in many cases, there are smaller numbers k that make $a^k \equiv 1 \pmod{n}$. For $\varphi(n)$, however, it is guarenteed for any a coprime to n that the congruence holds.

But, still, $\varphi(n)$ is not necessarily the smallest number that guarantees the congruence. In some cases, there is a smaller number that does the job and this number can be calculated by the *Carmichael function*, of which we have already used a part, when we discussed RSA.

The Carmichael function is usually denoted $\lambda(n)$ (but has nothing to do with the λ-calculus!). It is a bit difficult to give its definition in words. It is much easier, actually, to define it in Haskell:

```
lambda :: Natural → Natural
lambda 2 = tot 2
lambda 4 = tot 4
lambda n | twopower n    = (tot n) `div` 2
         | primepower n = tot n
         | even n ∧ primepower (n `div` 2) = tot n
         | otherwise     = let ps = map lambda (simplify $ trialfact n)
                           in foldl′ lcm 1 ps
  where simplify = map product ∘ group ∘ sort
```

There are two base cases stating that $\lambda(2)$ and $\lambda(4)$ equal $\varphi(2)$ and $\varphi(4)$ respectively. Otherwise, if n is a power of 2, then $\lambda(n)$ equals the half of $\varphi(n)$. An example is 8. We would expect that any group generated with coprimes of 8 has at most two members, since $\varphi(8) = 4$, and $\lambda(8) = 2$. We generate the groups with *map* (*generate* 8) (*coprimes* 8) and see:

```
[1]
[3,1]
[5,1]
[7,1].
```

The prediction, hence, is correct. We saw a similar result for 12, but that has other reasons as we will see below.

The function *twopower*, by the way, is defined as

```
twopower :: Natural → Bool
twopower 1 = True
twopower 2 = True
twopower n | even n     = twopower (n `div` 2)
           | otherwise = False
```

The next line states that for any primepower n, $\lambda(n) = \varphi(n)$.

The function *primepower* is defined as

```
primepower :: Natural → Bool
primepower n = length (nub $ trialfact n) ≡ 1
```

In other words, a primepower is a number whose prime factorisation consists of only one prime (which itself, however, may appear more than once). Since powers of 2 are handled in the previous guard, we are dealing here only with powers of odd primes. An example is 9, which is 3^2. The coprimes of 9 are 1, 2, 4, 5, 7 and 8. The totient number, hence, is 6. The groups are *map* (*generate* 9) (*coprimes* 9):

```
[1]
[2,4,8,7 5,1]
[4,7,1]
[5,7,8,4,2,1]
[7,4,1]
[8,1].
```

We see four different groups. The two trivial groups and two groups with 3 and 6 members respectively. Again, the prediction is correct.

The next line states that, if n is even and half of n is a primepower, then again $\lambda(n) = \varphi(n)$. An example is 18, since 18 is even and the half of 18 is 9, which is a power of 3. $\varphi(18)$ is 6, so we would expect to see groups with at most 6 elements. Here is the result for *map* (*generate* 18) (*coprimes* 18):

```
[1]
[5,7,17,13,11,1]
[7,13,1]
[11,13,17,7,5,1]
[13,7,1]
[17,1].
```

We see, again, four groups, the two trivial groups 1 and $n-1$ and two non-trivial groups with 3 and 6 members respectively.

We come to the *otherwise* guard. If n is not 2 or 4, not a power of 2 nor a power of another prime and not twice a power of a prime, then we do the following: we compute the factorisation, order the factors, group by equal factors, compute the primepower that corresponds to each group of factors and map λ on the resulting numbers. Then we compute the *lcm* of the results. In short: $\lambda(n)$ is the *lcm* of the λ mappend to the prime powers in the prime factorisation of n.

An example for a number that is not a primepower nor twice a primepower is 20. The factorisation of 20 is $\{2,2,5\}$. We compute the primepowers resulting in $\{4,5\}$. When we map λ on them, we should get $\lambda(4) = \varphi(4) = 2$ and $\lambda(5) = \varphi(5) = 4$. The *lcm* of 2 and 4 is 4 and, hence, $\lambda(20) = 4$. We, thus, should not see a group with more than 4 elements. We call *map* (*generate* 20) (*coprimes* 20) and see:

```
[1]
[3,9,7,1]
[7,9,3,1]
[9,1]
[11,1]
[13,9,17,1]
[17,9,13,1]
[19,1].
```

We see 6 different groups, the two trivial groups 1 and $n-1$ and four non-trivial groups with 2 and, respectively, 4 members.

The factorisation of 12 is $\{2, 2, 3\}$, so we apply λ on the numbers 4 and 3, which for both cases is 2. The *lcm* of 2 is just 2 and, therefore, we do not see groups with more than 2 members with the coprimes of 12.

Now, as you may have guessed, *Carmichael's theorem* states that, if a and n are coprime to each other, then

$$a^{\lambda(n)} \equiv 1 \pmod{n}. \tag{6.87}$$

For primes, the theorem is identical to Fermat's little theorem. For powers of odd primes, it reduces to Euler's theorem. The *lcm* of primepowers under the *otherwise*-guard is a consequence of the Chinese Remainder theorem and the very notion of the *lcm*. We know that, if $x \equiv 1 \pmod{n}$, then also $x \equiv 1 \pmod{mn}$. However, mn is not necessarily the first multiple of n and m that establishes the congruence. Any number that is a multiple of both, n and m, would have the same effect and the first number that is a multiple of both is $lcm(m, n)$.

The totient number of twice the power of an odd prime, $2p^k$, is the same as the totient number of that odd prime power, p^k: $\varphi(p^k) = \varphi(2p^k)$. The coprimes of p^k are all numbers from 1 to p^k that are not multiples of p, including all even numbers. Since twice that primepower is an even number, the even numbers are not part of the coprimes of that number. So, the coprimes of $2p^k$ in the range $1 \dots p^k$, are exactly half of the coprimes of p^k. But now, there are the coprimes in the second half $p^k \dots 2p^k$. Since the interval is the same in size and we eliminate the same number of numbers in that range as in the first half, namely the even numbers and the multiples of p, we end up with two sequences, each containing half as many numbers as the original sequence of coprimes of p^k. The two halfs together, therefore, make for the same amount of coprimes of p^k and $2p^k$. So, we can handle these cases in the same way.

The general rule, however, would produce the same result. According to the general rule, we would first compute λ for the individual primepowers and then the *lcm* of these values. The factorisation of a number that is twice a primepower contains the factor 2 and the primepower. The value for $\lambda(2)$ is $\varphi(2)$, which is 1. The *lcm* of 1 and another number is that other number. There, hence, is no difference between this rule and the general rule.

Now, what about the powers of 2 greater 4? To show that the greatest group of a power of 2 is half the totient of that number is quite an interesting exercise in group theory. The coprimes of a power of 2 have a quite peculiar structure, namely

$$1, \dots, m_1, m_2, \dots, n - 1.$$

Interesting are the middle numbers m_1 and m_2. They both are their own inverses, such that $m_1m_1 \equiv 1 \pmod{n}$ and $m_2m_2 \equiv 1 \pmod{n}$. The set of coprimes, therefore, consists of two symmetric halves, each starting and ending with a number that is its own inverse: $1 \dots m_1$ is the first half, $m_2 \dots n-1$ is the second half.

The number of coprimes is of course even since they consist of all odd numbers $1 \dots 2^k - 1$. Therefore, we do not have one central number, but the two middle numbers m_1 and m_2, which are one off the half of 2^k, that is $m_1 = 2^{k-1} - 1$ and $m_2 = 2^{k-1} + 1$. The following calculation shows that both m_1 and m_2 squared are immediately 1 modulo 2^k, for any 2^k with $k > 2$. For m_1 we have:

$$(2^{k-1} - 1)(2^{k-1} - 1).$$

When we multiply this out we get the terms $2^{k-1+k-1}$, which simplifies to 2^{2k-2}, $-2^{k-1} - 2^{k-1}$, which simplifies to -2^k, and 1:

$$2^{2k-2} - 2^k + 1.$$

We can factor 2^k out of the the first two terms:

$$2^k(2^{k-2} - 1) + 1,$$

and see clearly that the first remaining term is divided by 2^k and, thus, disappears modulo 2^k. We are left with 1 and this shows that m_1 is its own inverse.

For m_2, the proof is very similar, with the difference that we are left over with $2^k(2^{k-2}+1)$ for the first term. However, this term is a multiple of 2^k as well, and we are again left with 1.

Now, we can select a random generator of the group, say, a and look what it generates. For explicitness, we consider the case of $2^4 = 16$, whose coprimes are

$$1, 3, 5, 7, 9, 11, 13, 15$$

This group has the form (with arbitrary placement of the inverses of a and b):

$$1, a, b, m_1, m_2, b', a', n-1.$$

We see at once that no generator will create a sequence with 8 elements. Any sequence generated by exponentiation of a can contain only one of the elements m_1, m_2 and $n-1$. Since, if $a^k = m_2$, $a^{2k} = 1$ and, afterwards, the whole cycle repeats. If some a^l, for $k < l < 2k$, was m_1 or $n-1$, then a^{2l} would be 1 again. But that cannot be, since $2l > 2k$ and, therefore, a^{2l} is part of the second cycle, which has to be exactly the same as the first. But, obviously, there was only one 1 in the first cycle, namely at a^{2k} and there must be only one 1 in the second cycle, namely at a^{4k}. Therefore, there can be only one of the elements m_1, m_2 and $n-1$ in the sequence and this reduces the longest possible sequence for this example to 6, for instance:

1	2	3	4	5	6
a	b	m_2	b’	a’	1

Until here it looks fine. But observe that we now have one set with six numbers, the group generated by a, which we will call G, and its complement relative to the set of coprimes, which contains 2 elements. For the group above that containts m_2, the complement consists of m_1 and $n-1$.

Now, we will construct what is called a *coset*. A coset of G, in our context here, is a set of numbers resulting from one element of the complement of G, multiplied by all numbers of G. Let us say, this element is m_1. Then the coset of G created by m_1 denoted m_1G is

$$m_1G = \{m_1a, m_1b, m_1m_2, m_1b', m_1a', m_1\} \tag{6.88}$$

Note that this set contains six numbers. These numbers are necessarily different from all numbers in G, since the numbers in G form a group, the product of two members of which result in another member of it and, for each pair of members of the group c and d, there is one number x in the group, such that $xc = d$. m_1, however, is not member of the group and, if m_1 multiplied by c resulted in another member d, then we would have the impossible case that d is the result of multiplications of c for two different numbers: m_1 and x. Therefore, no number in m_1G can possibly equal any number in G.

But we do not have six numbers! We only have two numbers, namely m_1 and $n-1$. Therefore, no group that we create on a set of coprimes with such a structure can be greater than half of the number of coprimes. In our example that is four. With four numbers in G and four elements in the complement of G, we would have no problem at all. But, definitely, a group with six members does not work. □

A corollary of this simple but important argument is that the order of any subgroup of a group of coprimes must divide the number of coprimes. This extends the proof of Lagrange’s theorem for prime groups to composite groups. We will extend it even further in the future. For the moment, however, we can be satisfied with the result. We have proven Carmichael’s theorem.

6.9. How many Fractions?

How many negative numbers are there? That is a strange question! How do you want me to answer? I can tell you how many numbers of a specific kind are there only in relation to another kind of numbers. The words "how many" clearly indicate that the answer is again a number.

Let us try to state the question more pecisely: are there more or fewer negative numbers than natural numbers or are there exactly as many negative numbers as natural numbers?

To answer the question, I would like to suggest a way to compare two sets. To compare the size of two sets, A and B, we create a third set, which consists of pairs (a, b), such that $a \in A$ and $b \in B$. The sets are equal, if and only if every $a \in A$ appears exactly once in the new set and every $b \in B$ appears exactly once in the new set. If there is an $a \in A$ that does not appear in the new set, but all $b \in B$ appear exactly once, then B is greater than A. If there is a $b \in B$ that does not appear, but all $a \in A$ appear, then A is greater than B.

Furthermore, I suggest a way of counting a set A. We count a set by creating a new set that consists of pairs (a, n), such that $a \in A$ and $n \in \mathbb{N}$. For n, we start with 0 and, for each element in A, we increase n by 1 before we put it into the new set, like this:

```haskell
count :: [a] -> [(a, Natural)]
count = go 0
  where go _ [] = []
        go n (x:xs) = (x, n+1) : go (n+1) xs
```

The greatest number n, we find in the pairs of this set is the number of elements in A. Let us see if we can count the negative numbers in this manner. We count them by creating the set $\{(-1, 1), (-2, 2), (-3, 3), \dots\}$. Do we ever run out of negative or natural numbers? I don't think so. Should we ever feel that we run out of negative numbers, then we just take the current natural number and put a minus sign before it. Should we ever feel that we run out of natural numbers, then we simply take the current negative number and remove the negative sign. This proves, I guess, that there is a way to assign each negative number to exactly one natural number and vice versa. There are hence as many negative numbers as natural numbers.

Well, how many numbers are that? That are $|\mathbb{N}|$ numbers. If you want a word for that, call it *aleph-zero* and write it like this: $\aleph_0$. A set with this cardinality is infinite, but countable. Calling *count* on it, we will never get a final answer. But we will have a partial result at any given step.

What about all the integers? There should be twice as much as natural numbers, right? Let us see. We first create a set to count the natural numbers:

$$\{(1, 1), (2, 2), (3, 3), \dots\}$$

Then we insert a negative number before or behind each positive number:

$$\{(1,1),(-1,2),(2,3),(-2,4),(3,5),(-3,6),\dots\}$$

Again, it appears that we do not run out of natural numbers to count all the integers. The set of integers, hence, has cardinality $\aleph_0$ too.

What about fractions? On the first sight, fractions look very different. There are infinitely many of them between any two natural numbers as we have seen with Zeno's paradox. But now comes Cantor and his first diagonal argument to show that fractions are countable and, therefore, that the set of fractions has cardinality $\aleph_0$.

Cantor's proof, his first diagonal argument, goes as follows. He arranged the fractions in a table, such that the first column contained the integers starting from 1 in the first row and counting up advancing row by row. The integers correspond to fractions with 1 as denominator. So, we could say, the first column of this table is dedicated to denominator 1. The second column, correspondingly, is dedicated to denominator 2; the third to denominator 3 and so on. Then, the rows are dedicated likewise to numerators. The first row contains numerator 1, the second contains numerator 2, the third numerator 3 and so on. Like this:

$\frac{1}{1}$	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\dots$
$\frac{2}{1}$	$\frac{2}{2}$	$\frac{2}{3}$	$\frac{2}{4}$	$\frac{2}{5}$	$\frac{2}{6}$	$\dots$
$\frac{3}{1}$	$\frac{3}{2}$	$\frac{3}{3}$	$\frac{3}{4}$	$\frac{3}{5}$	$\frac{3}{6}$	$\dots$
$\frac{4}{1}$	$\frac{4}{2}$	$\frac{4}{3}$	$\frac{4}{4}$	$\frac{4}{5}$	$\frac{4}{6}$	$\dots$
$\frac{5}{1}$	$\frac{5}{2}$	$\frac{5}{3}$	$\frac{5}{4}$	$\frac{5}{5}$	$\frac{5}{6}$	$\dots$
$\frac{6}{1}$	$\frac{6}{2}$	$\frac{6}{3}$	$\frac{6}{4}$	$\frac{6}{5}$	$\frac{6}{6}$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

This table is a brute-force approach to list all fractions in two dimensions. Obviously, this table contains all possible fractions, since, for any possible pair of numbers, it contains a fraction, which, however, is not necessarily in canonical form, so the table contains duplicates.

Then, using this table, he created a sequence. He started with the first cell containing $\frac{1}{1}$. From there he went to the next row $\frac{2}{1}$. Then he applied the rule *up*, that is he went up following a diagonal line to the first row, so he would eventually reach $\frac{1}{2}$. He went one to the right and then applied rule *down*, that is he went down following a diagonal line to the first column, so he would eventually reach $\frac{3}{1}$. Now he continued to the next row and repeated the process of going *up* and *down* in diagonal lines infitely adding the number of each cell he crossed to the sequence.

The sequence evolves as follows: Cantor starts with $\frac{1}{1}$, adds $\frac{2}{1}$, applies rule *up* and adds $\frac{1}{2}$, goes to the right, adds $\frac{1}{3}$ and applies rule *down* adding $\frac{2}{2}$; then he goes to the next row adding $\frac{4}{1}$ and goes *up* again adding $\frac{3}{2}$, $\frac{2}{3}$ and $\frac{1}{4}$ and so he goes on forever.

We can reformulate this rule in Haskell, which will make the process clearer:

```haskell
cantor1 :: [Ratio]
cantor1 = (1 % 1) : go 2 1
  where go   n 1 = up   n 1 ++ go 1 (n + 1)
        go   1 d = down 1 d ++ go (d + 1) 1
        down n 1 = [n % 1]
        down n d = (n % d) : down (n + 1) (d - 1)
        up   1 d = [1 % d]
        up   n d = (n % d) : up   (n - 1) (d + 1)
```

When we look at rule *up*, starting at the bottom of the code, we see the base case where the numerator is 1. In this case, we just yield $[1 \% d]$. Otherwise, we call *up* again with the numerator n decremented by 1 and the denominator incremented by 1. *up* 4 1, thus, is processed as follows:

```haskell
up 4 1 = (4 % 1) : up 3 2
up 3 2 = (3 % 2) : up 2 3
up 2 3 = (2 % 3) : up 1 4
up 1 4 = [1 % 4]
```

yielding the sequence $\frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}$. *go*, after calling *up*, proceeds with *go* 1 $(n+1)$. We, hence, would continue with *go* 1 5, which calls *down*. The base case of *down* is the case where the denominator is 1. Otherwise, we increment the numerator by 1 and decrement the denominator by 1. *down* 1 5, hence, is processed as follows:

```haskell
down 1 5 = (1 % 5) : down 2 4
down 2 4 = (2 % 4) : down 3 3
down 3 3 = (3 % 3) : down 4 2
down 4 2 = (4 % 2) : down 5 1
down 5 1 = [5 % 1]
```

yielding the sequence $\frac{1}{5}, \frac{2}{4}, \frac{3}{3}, \frac{4}{2}, \frac{5}{1}$. When we put the sequence together, including the first steps, we see

$$\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{1}{3}, \frac{2}{2}, \frac{3}{1}, \frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, \frac{1}{5}, \frac{2}{4}, \frac{3}{3}, \frac{4}{2}, \frac{5}{1}, \dots$$

When we reduce all fractions to canonical form, we see a lot of repetitions:

$$\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{1}, \frac{3}{1}, \frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{5}{1}, \dots$$

We see the numbers $\frac{1}{1} = 1$, $\frac{2}{1} = 2$ and $\frac{1}{2}$ repeated several times. They will continue to appear over and over again and, even worse, other numbers will start to reappear too. That is, before we can use this sequence to count fractions, we need to filter duplicates out leading to the sequence

$$\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{1}{3}, \frac{3}{1}, \frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, \frac{1}{5}, \frac{5}{1}, \dots$$

But now we see that we can enumerate, that is count, the fractions creating the sequence

$$\left(\frac{1}{1}, 1\right), \left(\frac{2}{1}, 2\right), \left(\frac{1}{2}, 3\right), \left(\frac{1}{3}, 4\right), \left(\frac{3}{1}, 5\right), \left(\frac{4}{1}, 6\right), \left(\frac{3}{2}, 7\right), \left(\frac{2}{3}, 8\right), \left(\frac{1}{4}, 9\right), \dots$$

This clearly shows that the cardinality of the set of fractions is $\aleph_0$. □

This result may feel a bit odd on the first sight. We clearly have the feeling that there must be more fractions than integers, because, between any two integers, there are infinitely many fractions. When we think of a visualisation with a pair of balances, with the fractions being in one balance and the integers in the other, then, what we would see at any given instance, clearly indicates that there must be more fractions than integers. However, our feeling betrays us, when it comes to infinity. Indeed, our feeling was not made for infinity. Therefore, at least if we accept the notions of comparison and counting outlined above, then we have to accept the result of Cantor's argument. Even further, I would say that the fact that this argument shows things in a way that contradicts our spontaneous way to see these things, underlines the extraordinary quality of this argument. Cantor lets us see things that are usually hidden from our perception. This makes Cantor, who was seen by his contemporary opponents as a kind of sorcerer, a true magus.

It is, by the way, quite simple to extend the argument to negative fractions. We just have to insert behind each number its additive inverse, resulting in the sequence:

$$\left(\frac{1}{1}, 1\right), \left(-\frac{1}{1}, 2\right), \left(\frac{2}{1}, 3\right), \left(-\frac{2}{1}, 4\right), \left(\frac{1}{2}, 5\right), \left(-\frac{1}{2}, 6\right), \left(\frac{1}{3}, 7\right), \left(-\frac{1}{3}, 8\right), \dots$$

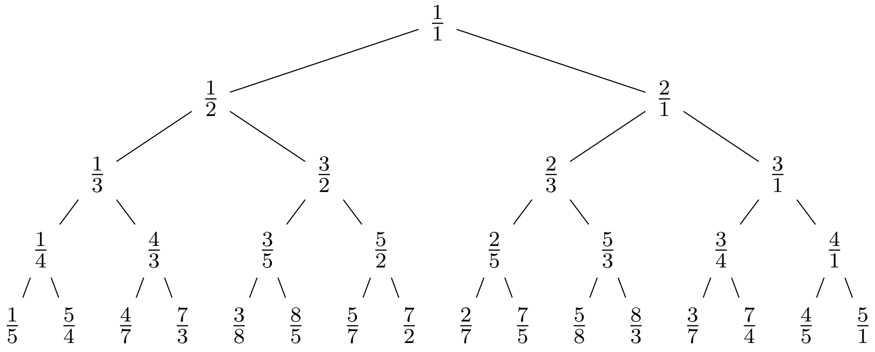
Indeed, there appears to be a lot of room for new numbers, once we are dealing with infinity. This led the great David Hilbert to the analogy of the hotel, which is today named after him *Hilbert's Hotel*. In this analogy, there is a hotel with the uncommon property of having infinitely many rooms. This hotel will never run out of rooms for new guests. If a new guest arrives and there are already infinitely many guests, the manager just asks the guests to move one room up, *i.e.* the guest currently in room 1 moves to room 2, the guest in room 2 moves to room 3 and so on. At the end of the process, room 1 is free for the new arriving guest. Since there are infinitely many rooms, there is no guest, even though there are infinitely many of them already, who would not find a room with a room number one greater than his previous room number.

This approach works for any number of finitely many guests arriving. But even when infinitely many new guests arrive, the manager, still, has resources. In this case, he could ask the guests to move to a room with a number twice the number of his current room, *e.g.* the guest in room 1 would move to room 2, the guest in room 2 would move to room 4, the guest in room 3 would move to room 6 and so on leaving infinitely many rooms with odd room numbers unoccupied and making room for infinitely many new guests.

But let us go back to more technical stuff. In spite of its ingenuity, Cantor's argument is not perfect. In particular, the sequence and how it is created is quite uggly, but, apparently, nobody, for more than hundred years, cared too much about that. Then, in 2000, Neil Calkin and Herbert Wilf published a paper with a new sequence with a bunch of interesting properties that make this sequence for enumerating the fractions much more attractive than Cantor's original sequence. The beginning of the sequence is

$$\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, \frac{4}{1}, \dots$$

The sequence, as we will show in a minute, corresponds to a *binary tree* of the form



When you have a closer look at the tree, you see that the kids of each node are created by a simple formula. If the current node has the form $\frac{n}{d}$, then the left kid corresponds to $\frac{n}{n+d}$ and the right kid corresponds to $\frac{n+d}{d}$. For instance, the kids of $\frac{1}{1}$ are $\frac{1}{1+1}$ and $\frac{1+1}{1}$. The kids of $\frac{3}{2}$ are $\frac{3}{3+2}$ and $\frac{3+2}{2}$.

We can easily create this tree in Haskell. First, we need a data type to represent the tree:

```
type CalWiTree = Tree Ratio
```

The *Tree* data type is in fact not a binary tree, but a generic tree with an arbitrary number of nodes. But it is simple to implement and serves our purpose. The data type is parametrised, so we define a specialised data type *Tree Ratio* and the type synonym *CalWiTree* referring to this type. Now we create the tree with:

```
calWiTree :: Zahl → Ratio → CalWiTree
calWiTree 1 r          = Node r [ ]
calWiTree i r@(Q n d)  = Node r [calWiTree (i − 1) (n % (n + d)),
                                 calWiTree (i − 1) ((n + d) % d)]
```

The function takes two arguments, a *Zahl* and a *Ratio*. The *Ratio* is the starting point and the *Zahl* is the number of generations we want to create. Often we do not want the function to create the entire sequence – for that a lot of patience and memory resources would be necessary – but only a tiny part of it. In this case, we set the *Zahl* to $i \geq 1$. If i reaches 1, we create the current node without kids. If we want the function to create the entire infinite tree, we just assign a value $i < 1$. If $i \neq 1$, we create a node with r as data and the kids resulting from calling *calWiTree* on $\frac{n}{n+d}$ and $\frac{n+d}{d}$ with i decremented by 1.

This shows that there is a very simple algorithm to generate the tree. We will now show that Calkin-Wilf tree and Calkin-Wilf sequence are equivalent. We do so by creating an algorithm that converts the tree to the sequence.

We may be tempted to do this with a typical recursive function that does something with the current node and then adds the result of the operation to the partial sequences that result from recursively calling the function on the left and the right kid. This approach, however, is *depth-first*. The resulting sequences would follow the branches of the tree. It would create partial sequences like, for instance, $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$ But what we need is partial sequences that cover generation by generation, *i.e.* $\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}$ and so on. In other words, we need a *breadth-first* approach.

Since this is a generic problem, we can define a function on the level of the generic *Tree* data type that creates a sequence composed of subsequences corresponding to tree generations:

```
getKids :: Natural → Tree a → [a]
getKids 1 (Node r _)         = [r]
getKids n (Node r [ ])       = [ ]
getKids n (Node r (x : xs))  = getKids (n − 1) x ++
                               getKids n (Node r xs)
```

The function receives two arguments, a *Natural*, which determines the generation we want to obtain, and the tree on which we are operating. If the generation is 1, we just give back the data of the current node. Otherwise, we distinguish two cases: If the current node has no kids, then the result is the empty list. This indicates that we have exhausted the current (finite) tree. Otherwise, we advance recursively on the head and tail of the list of kids. Decisive is that we do not add anything to the resulting sequence, before we have reached the intended depth n. This way, the function produces a sequence containing all kids on level n. We now just apply *getKids* to all generations in the tree:

```
calWiTree2Seq :: CalWiTree → [Ratio]
calWiTree2Seq t = go 1
  where go n = case getKids n t of
                 [] → []
                 sq → sq ++ go (n + 1)
```

We have shown that there is a simple algorithm to generate the tree and that there is a simple algorithm to convert the tree into the sequence. The latter is quite useful, since it means that tree and sequence are equivalent. This allows us to prove some crucial properties of the sequence using the tree, which is much simpler than proving them on the sequence directly.

Already a quick glance at the tree reveales some interesting properties, for instance, that, in all cases, the left kid is smaller and the right kid is greater than the parent node. This, of course, is just a consquence of the generating algorithm. We also see that the integers are all in the right-most branch, which equals the first column in Cantor's table. The left-most branch equals the first row in Cantor's table: it contains all fractions with 1 in the numerator. We also see that all fractions are in canonical form, different from Cantor's table. Also, no fraction repeats and, as far as we can tell, the fractions appear to be complete.

The crucial properties, those that we need to show that the Calkin-Wilf sequence contains exactly all fractions and, thus, can be used for Cantor's argument in place of the old sequence, are:

1. All fractions in the tree are in canonical form;
2. Every possible fraction (in canoncial form) appears in the tree;
3. Every fraction appears exactly once.

The first property is simple to prove. We observe that the first fractions appearing in the tree that do not have 1 in numerator or denominator have there either 2 or 3. 2 and 3 a coprime to each other. Summing two coprimes, a and b, will lead to a number that again is coprime to both a and b. Therefore, if we have at a node n a fraction whose numerator and denominator are coprime to each other, the whole subtree below n will contain fractions whose numerator and denominator are coprime to each other. Therefore, the subtrees below $\frac{3}{2}$ and $\frac{2}{3}$ will only contain fractions in canonical form.

The fractions that actually have 1 in numerator or denominator, however, will necessarily lead to a fraction whose numerator and denominator are coprime to each other, since no number n, except $n = 1$, shares any divisor with $n + 1$. Since we start the tree with $\frac{1}{1}$, the whole tree can only contain fractions in canonical form. □

We prove the second property by contradiction. Let us assume that there are fractions that do not appear in the tree. Then, there is a number of fractions that have the smallest denominator and, among those, there is one with the smallest numerator. Let $\frac{n}{d}$ be this fraction. If $n > d$, then $\frac{n-d}{d}$ cannot appear either, since $\frac{n}{d}$ would be its right kid. But that is a contradiction to our assumption that $\frac{n}{d}$ was the nonappearing fraction with the smallest numerator among those fractions with denominator d, but, obviously, $n - d$ is an even smaller numerator. If $n < d$, then $\frac{n}{d-n}$ cannot appear either, since, $\frac{n}{d}$ would be its left kid. But that again is a contradiction, since the denominator is smaller than the denominator of one of the fractions we assumed to be those with the smallest denominator. The only way to solve this dilemma is to assume that n and d are equal. Then, indeed, $\frac{n-d}{d} = 0$ and $\frac{n}{n-d} = \perp$ would not be in the tree. But such a fraction with $n = d$ is irrelevant, since it reduces to $\frac{1}{1}$, which already is in the tree. $\square$

To prove the third property, we first observe that 1 is the root of the tree. Since any fraction below 1 is either $\frac{n}{n+d}$ or $\frac{n+d}{d}$, there cannot be a fraction with $n = d$. With this out of the way, we can argue as we did for the second property: we assume there are fractions that appear more than once. From all these fractions, there is a group that shares the smallest denominator and, among this group, one with the smallest numerator. But this fraction is then either the left kid of two fractions of the form $\frac{n}{d-n}$, making the denominator of these fractions even smaller, or the right kid of two fractions of the form $\frac{n-d}{d}$, making the numerator of these fractions even smaller. In both cases we arrive at a contradiction. $\square$

We can also prove directly – but the argument may be more subtle or, which is the same, almost self-evident. We know that all nodes are of either of the forms $\frac{n}{n+d}$ or $\frac{n+d}{d}$. Since, except for the root node, we never have $n = d$, no node derived from one of those fractions can ever equal one derived from the other. To say that two fractions derived from such nodes are equal, would mean that we could have two numbers n and d, such that $n = n + d$ and $d = n + d$. That would only work if $n = 0$ and $d = 0$. But that case cannot occur. $\square$

That taken all together shows that the Calkin-Wilf sequence contains all fractions exactly once. Since we can enumerate this sequence, we can enumerate all fractions and, hence, $|\mathbb{Q}| = \aleph_0$. $\square$

There is still another advantage of this sequence over Cantor's original one. There is a simple, yet quite exciting way to compute which is the n^{th} fraction. The key is to realise that we are dealing with a structure, namely a binary tree, that stores sequences of binary decisions. At any given node in the tree, we can go either right or left. We could, therefore, describe the n^{th} position as a trajectory through the tree, where at each node, we take a binary decision: going right or going left. An efficient way to encode a sequence of binary decisions is a binary number, and, indeed, the position in the sequence, represented as a binary number leads to the fraction at that position. Here is a function that, given a natural number n, returns the fraction in the Calkin-Wilf sequence at position n:

```
calwiR :: Natural → Ratio
calwiR = go (0 % 1) ∘ toBinary
  where go r []                = r
        go r@(Q n d) (0 : xs) = go (n % (n + d)) xs
        go r@(Q n d) (1 : xs) = go ((n + d) % d) xs
```

We start by converting n to binary format (*i.e.* a list of 0s and 1s). Then we call *go* starting with 0, since, for the first number 1, we want position 1. If we have exhausted the binary number, the result is just r, the rational number we pass to *go*. Otherwise, we distinguish two cases: the head of the binary number being 0 or 1. If it is 0, we follow the left branch, which has the fraction $\frac{n}{n+d}$ at its top; if it is 1, we follow the right branch with the fraction $\frac{n+d}{d}$.

Consider $n = 25$ as an example. 25 in binary format is 11001. We go through the steps:

```
go (Q 0 1) [1,1,0,0,1] = go (1 % 1) [1,0,0,1]
go (Q 1 1) [1,0,0,1] = go (2 % 1) [0,0,1]
go (Q 2 1) [0,0,1] = go (2 % 3) [0,1]
go (Q 2 3) [0,1] = go (2 % 5) [1]
go (Q 2 5) [1] = go (7 % 5) []
go (Q 7 5) [] = Q 7 5
```

Let us check if this is true; `take 1 $ drop 24 $ calWiTree2Seq (calWiTree 5 (1 % 1))` gives

`[Q 7 5]`,

which corresponds to the correct result $\frac{7}{5}$. With this we can create the sequence much simpler without deviating through the tree:

```
calwis :: [Ratio]
calwis = map calwiR [1 ..]
```

We can also do the opposite: compute the position of any given fraction. For this, we just have to turn the logic described above around:

```
calwiP :: Ratio → Natural
calwiP = fromBinary ∘ reverse ∘ go
  where go (Q 0 _)  = []
        go (Q n d) | n ⩾ d      = 1 : go ((n − d) % d)
                   | otherwise = 0 : go (n % (d − n))
```

We look at the fraction and, if $n \geq d$, then we add 1 to the digits of the resulting binary number, otherwise, we add 0. Note that there is only one case where $n = d$ (as we have proven above), namely the root node of the tree. In all other cases, we either have $n > d$ or $n < d$. In the first case, we know that the fraction is a right kid and in the second case, we know it is a left kid. When we call this function on $\frac{7}{5}$, we see the steps:

```
go (Q 7 5) = 1 : go (2 % 5)
go (Q 2 5) = 0 : go (2 % 3)
go (Q 2 3) = 0 : go (2 % 1)
go (Q 2 1) = 1 : go (1 % 1)
go (Q 1 1) = 1 : go (0 % 1)
go (Q 0 1) = [],
```

which leads to the list 1 : 0 : 0 : 1 : 1 : [ ]. This evaluated and reversed is 1, 1, 0, 0, 1, which, converted to decimal representation, is 25.

Surprisingly or not, the Calkin-Wilf sequence is not completely new. A part of it was already studied in the 19^{th} century by German mathematician Moritz Stern (1807 – 1894), successor of the early deceased sucessor of Gauss at the University of Göttingen, Lejeune-Dirichlet, and professor of Bernhard Riemann. The numerators of the Calkin-Wilf sequence correspond to *Stern's diatomic sequence*. Using the Calkin-Wilf sequence, we can produce Stern's sequence with the function

```
stern :: [Natural]
stern = map numerator calwis
  where numerator (Q n _) = n
```

take 32 *stern* shows:

1, 1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, 1, 5, 4, 7, 3, 8, 5, 7, 2, 7, 5, 8, 3, 7, 4, 5, 1.

Mapping *denominator* defined as *denominator* (*Q* _ *d*) = *d* on the Calkin-Wilf sequence would give a very similar result: the Stern sequence one ahead, *i.e.*:

1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, 1, 5, 4, 7, 3, 8, 5, 7, 2, 7, 5, 8, 3, 7, 4, 5, 1, 6.

Edsgar Dijkstra, the great pioneer of the art of computer programming, studied this sequence not knowing that it had already been studied before. He called it the *fusc* sequence and generated it with the function

```
fusc :: Natural -> Natural
fusc 0 = 0
fusc 1 = 1
fusc n | even n    = fusc (n `div` 2)
       | otherwise = let k = (n - 1) `div` 2
                     in fusc k + fusc (k + 1)
```

From the definition of the *fusc* function, we can read some of the many properties of Stern's sequence (and the Calkin-Wilf tree). First, an even number has the same value as half of that number, for instance *fusc* 3 is 2 and so is *fusc* 6, *fusc* 12, *fusc* 24 and so on. Even numbers in binary format end on zeros. For instance, 3 in binary notation is 11. 2 × 3 is 110, 2 × 6 is 1100, 2 × 12 is 11000 and so on. The binary format clearly indicates that, after having reached the number before the trail of zeros at the end, we go down in a straight line following the left branch of that node in the Calkin-Wilf tree. Since, in the left path, the numerator never changes, the result of $fusc(n)$ equals the result of $fusc(2n)$.

We also see that for any power of 2, *fusc* equals *fusc* 1, which is 1; *map fusc* $[2, 4, 8, 16, 32, 64, 128, 256]$, hence, gives $[1, 1, 1, 1, 1, 1, 1, 1]$. Note that, looking at the Calkin-Wilf tree, this is immediate obvious, since powers of 2 in binary representation are numbers of the form $1, 10, 100, 1000, \dots$ Those numbers indicate that we navigate through the tree in a straight line following the left branch of the root node $\frac{1}{1}$.

The *fusc* results of powers of two minus one $(1, 11, 111, 1111, \dots)$ equal the number of digits of this number in binary form. This is the right outer branch of the tree with the integers.

The *fusc* results of powers of two plus one $(1, 11, 101, 1001, \dots)$ also equal the number of digits in the binary representation of that number. These numerators appear in the immediate neighbours of the powers of two in the left outer branch of the tree, for instance $\frac{3}{2}, \frac{4}{3}, \frac{5}{4}, \frac{6}{5}, \dots$

What about numbers with an alternating sequence of 1s and 0s, like 101010101? Those numbers are not in the outer branches and not even close to them. Indeed, they tend to the horizontal centre of the tree. The first 1 leads to node $\frac{1}{1}$. We now go left, that is, we add the numerator to the denominator leading to $\frac{1}{2}$; we then add the denominator to the numerator leading to $\frac{3}{2}$; then we add the numerator to the denominator again leading to $\frac{3}{5}$ and so we go on and obtain the fractions $\frac{8}{5}, \frac{8}{13}, \frac{21}{13}, \frac{21}{34}, \frac{55}{34}, \dots$ Do you see the point? All the numerators and denominators are Fibonacci numbers! Well, what we did above, adding the two numbers we obtained before starting with the pair $(1, 1)$, is just the recipe to create the Fibonacci numbers.

An amazing property of *fusc*, found by Dijkstra, is the fact that two numbers whose binary representations are the reverse of each other have the same *fusc* result. 25, for instance, is 11001. The reverse, 10011, is 19, and *fusc* $19 =$ *fusc* $25 = 7$.

For the Calwin-Wilf tree this means that, when we have two trajectories through the tree, where each step after the root node is the opposite of the simultaneous step of the other one, we arrive at two fractions with the same numerator. The trajectory defined by the binary sequence $1, 1, 0, 0, 1$ leads, first, to the root node $\frac{1}{1}$, then through $\frac{2}{1}$, $\frac{2}{3}$ and $\frac{2}{5}$ to $\frac{7}{5}$. The reverse of this sequence, $1, 0, 0, 1, 1$ leads, first, to the root node $\frac{1}{1}$ and then through $\frac{1}{2}$, $\frac{1}{3}$ and $\frac{4}{3}$ to $\frac{7}{3}$.

A similar is true for two numbers, whose binary sequences can be transformed into one another by inverting the inner bits. For instance, 11001, 25, can be transformed into 10111, inverting all bits, but the first and the last one. 10111 is 23 and *fusc* $19 =$ *fusc* $23 =$ *fusc* 25. What about the bit inverse of 19? That is 11101, the reverse of 10111 and 29 in decimal notation. Therefore *fusc* $19 =$ *fusc* $23 =$ *fusc* $25 =$ *fusc* 29.

We can reformulate this result in terms of group theory. We have three basic transformations i, the identity, ρ, the reverse, and β, the bit inverse. We add one more transformation, the composition of ρ and β and call it $\sigma = \rho \cdot \beta$. The operation defined over this set is composition. We see that the identity is part of the set; for each transformation, its inverse is in the set, too, because $\rho \cdot \rho = i$, $\beta \cdot \beta = i$ and $\sigma \cdot \sigma = i$. To illustrate the logic of this group with the numbers above, we define it on the base string 19, which is 10011:

$i = 10011$
$\rho = 11001$
$\beta = 11101$
$\sigma = 10111.$

Now we can play around and see that we will never generate a string that is not already in the group:

$\rho \cdot i = 11001$
$\rho \cdot \rho = i = 10011$
$\beta \cdot \beta = i = 10011$
$\sigma \cdot \sigma = i = 10011$
$\rho \cdot \beta = \sigma = 10111$
$\beta \cdot \rho = \sigma = 10111$
$\sigma \cdot \rho = \beta = 11101$
$\sigma \cdot \beta = \rho = 11001$
...

All elements of one group are in the same generation of the Calkin-Wilf tree, since they all have the same number of digits. Numbers with a symmetric binary representation, such that $\rho = i$, lead to groups with only two distinguishable members, for instance $fusc(2^n+1) = fusc(2^{n+1}-1)$. The same is true for numbers with a binary representation such that $\rho = \beta$, for instance, 101011 (43) = 110101 (53) and $fusc(43) = fusc(53) = 13$.

There are infinitely many numbers with the same *fusc* result. Most of these numbers have trailing zeros and, as such, are in the long shadow thrown by one of the original odd numbers with the same result. One example of such a shadow is the outer left branch, which maintains the numerator of the root node $\frac{1}{1}$ and also maintains the leading 1 in the binary representation of its positions, merely adding more and more 0s to it.

How many "original" numbers in this sense are there for a given *fusc* result n? The answer is simple if we consider two facts: 1. The fractions in the Calkin-Wilf tree are in canonical form, *i.e.* numerator and denominator do not share divisors, and 2. Any position number whose binary representation ends with 1, points to a right kid and, for all fractions $\frac{n}{d}$ that are right kids: $n > d$. Binary numbers, however, that end with 0, point to a left kid and, therefore, $n < d$. In other words, the number of original numbers for a given numerator n is $\varphi(n)$, the totient number of n. The denominators of the original fractions are the coprimes of n.

The numerator 7, for instance, appears in six positions: 19, 23, 25, 29, 65 and 127. The denominators of the fractions at those positions are 3, 2, 5, 4, 6 and 1. For numerator 8, there are only four such numbers: 21, 27, 129 and 255. The denominators at those positions are 5, 3, 7 and 1. Note that 21 in binary format is 10101, which is its own reverse, and 27 is the bit inverse of 21, namely 11011, which also is its own reverse.

Furthermore, those numbers appear in groups with 2 or 4 members, depending on the properties of the binary representation. The number of such groups, hence, is $\frac{\varphi(n)}{k}$, where k is some integer that divides $\varphi(n)$. For 7: $k = 3$, since there are two groups, one containing 4 elements, the other containing 2. For 8: $k = 2$, since there are two groups, both containing 2 elements.

The last group is the one consisting of binary numbers with n bits, *i.e.* $2^{n-1}+1$ and 2^n-1. The other groups appear in generations of the Calkin-Wilf tree before the generation with that final group. For 7, the generation of the group with four members is the fifth generation and the generation with the final group is of course the seventh generation. In other cases, the groups can be many generations apart. The numerator 55, for instance, appears for the first time in generation 10, namely in the fraction $\frac{55}{34}$ (both Fibonacci numbers). This is far off from generation 55 with the group consisting of $\frac{55}{1}$ and $\frac{55}{54}$.

Interestingly, Dijkstra was not aware of the relation of the *fusc* algorithm to the Stern sequence, and the Calkin-Wilf tree was not even around at that time. Dijkstra describes *fusc* as a state automaton that parses strings consisting of 1s and 0s. The parsing result would be a number, namely the result of *fusc*. We could now say that the Calkin-Wilf tree is a model that gives meaning to the strings in terms of trajectories through the tree.

A final remark relates to the product of one generation in the tree. Each generation consists of fractions whose numerators and denominators were created by adding the numerators and denominators of the fractions of the previous generation. We start with the fraction $\frac{1}{1}$. In consequence, in any generation, there is for any fraction $\frac{n}{d}$ a fraction $\frac{d}{n}$. The fractions in the fifth generation for example, the one containing the fractions at positions 19, 23, 25 and 29 in the Calkin-Wilf sequence, can be paired up in the following way:

$$\left(\frac{1}{5},\frac{5}{1}\right),\left(\frac{5}{4},\frac{4}{5}\right),\left(\frac{4}{7},\frac{7}{4}\right),\left(\frac{7}{3},\frac{3}{7}\right),\left(\frac{3}{8},\frac{8}{3}\right),\left(\frac{8}{5},\frac{5}{8}\right),\left(\frac{5}{7},\frac{7}{5}\right),\left(\frac{7}{2},\frac{2}{7}\right).$$

The product of each pair is 1. The product of all fractions in one generation is therefore 1 as well. You can try this out with the simple function

```
genprod :: Natural → CalWiTree → Ratio
genprod n t = product (getKids n t)
```

A sequence with so many nice properties, one might feel to say with some poetic fervour, cannot be meaningless. Isn't there anything in the (more or less) real world that these numbers would actually count? It turns out there is. There are in fact many things the Stern sequence actually counts. Just to mention two things: It counts odd binomial coefficients of the form $\binom{n-r}{r}, 0 \leq 2r \leq n$. That is the odd numbers in the first half of the lines $n - r$ in Pascal's triangle. This would translate to a function creating a sequence of numbers of the form

```
oddCos :: Natural → [Natural]
oddCos n = filter odd [choose (n − r) r | r ← [0 . . (n ‘div‘ 2)]]
```

fusc $(n + 1)$ is exactly the size of that sequence. For instance: *oddCos* 5 is $[1, 3]$ and *fusc* 6 is 2; *oddCos* 6 is $[1, 5, 1]$ and *fusc* 7 is 3; *oddCos* 7 is $[1]$ and *fusc* 8, which is a power of 2, is 1; *oddCos* 8 is $[1, 7, 15, 1]$ and *fusc* 9 is 4; *oddCos* 9 is $[1, 21, 5]$ and *fusc* 10 is 3.

Moritz Stern arrived at his sequence, when studying ways to represent numbers as powers of 2. Any number can be written as such a sum and most numbers even in various ways. For instance, $2 = 2^0 + 2^0$, $3 = 2^0 + 2^1 = 2^0 + 2^0 + 2^0$, $4 = 2^1 + 2^1 = 2^0 + 2^0 + 2^1$, $5 = 2^0 + 2^2 = 2^0 + 2^1 + 2^1$ and so on. Stern focussed on so called *hyperbinary* systems, that is sums of powers of 2, where any power appears at most twice. For instance, $3 = 2^0 + 2^1$ is such a system, but $3 = 2^0 + 2^0 + 2^0$ is not. Stern's sequence counts the number of ways this is possible for any number $n - 1$. In other words, *fusc* $(n + 1)$ is exactly the number of hyperbinary systems for n. For 3, as an example, there is only one way and *fusc* 4 is 1; for 4, there are 3 such systems: $2^0 + 2^0 + 2^1$, $2^1 + 2^1$ and 2^2 and *fusc* 5 is 3; for 5, there are only 2 such systems: $2^0 + 2^1 + 2^1$ and $2^0 + 2^2$ and *fusc* 6 is 2.

Finding all hyperbinary systems for a number n is quite an interesting problem in its own right. A brute-force and, hence, inefficient algorithm could apply the following logic. We first find all powers of 2 less than or equal to n:

```
pows2 :: Natural → [Natural]
pows2 n = takeWhile (⩽ n) [2 ↑ x | x ← [0 . .]]
```

We then create all permutations of this set and try to build sums that equal n:

```
hyperbin :: Natural → [[Natural]]
hyperbin n = nub (go $ perms $ pows2 n)
  where go pss = filter (λk → sum k ≡ n)
                        [sort (sums 0 ps ps) | ps ← pss]
        sums _ [ ] [ ]        = [ ]
        sums s [ ] ds         = sums s ds [ ]
        sums s (p : ps) ds | s + p > n = sums s ds ps
                           | s + p ≡ n = [p]
                           | otherwise = p : sums (s + p) ps ds
```

Note that we pass the available pool of powers of $2 \leq n$ twice to *sums*. When the first instance is exhausted or $s + p > n$, we start to use the second instance of the pool. This reflects the fact that we are allowed to use every number twice. If the sum $s + p$ equals n, we have found a valid hyperbinary system. Otherwise, if $s + p < n$, we continue adding the current power to the result set. In *go*, we try *sums* on all permutations of the powers filtering the resulting sets for which *sum* equals n. We sort each list to make lists with equal elements equal and to, thus, be able to recognise duplicates and to remove them with *nub*.

7. Real Numbers

7.1. $\sqrt{2}$

Until now, we have looked at *discrete* numbers, that is numbers that are nicely separated from each other so that we can write them down unmistakebly and always know of which number we are currently talking. Now we enter a completely different universe. The universe of continuous numbers that cannot be written down in a finite number of steps. The representation of these numbers consists of infinitely many elements and, therefore, we will never be able to write the number down completely with all its elements. We may give a finite formula that describes how to compute the specific number, but we will never see the whole number written down. Those numbers are known since antiquity and, apparently, their existence came as a great surprise to Greek mathematicians.

The first step of our investigations into this kind of numbers, is to show that they *exist*, *i.e.*, there are contexts where they arise naturally. To start, we will assume that they are not necessary. We assume that all numbers are either natural, integral or fractional. Indeed, any of the fundamental arithmetic operations, $+, -, \times$ and $/$, applied on two rational numbers results always in a rational number, *i.e.* an integer or a fraction. We could therefore suspect that the result of any operation is a rational number, *i.e.* an integer or a fraction.

What about $\sqrt{2}$, the square root of 2? Let us assume that $\sqrt{2}$ is as well a fraction. Then we have two integers n and d, coprime to each other, such that

$$\sqrt{2} = \frac{n}{d} \tag{7.1}$$

and

$$2 = \left(\frac{n}{d}\right)^2 = \frac{n^2}{d^2}. \tag{7.2}$$

Any number can be represented as a product of primes. If $p_1 p_2 \dots p_n$ is the prime factorsiation of n and $q_1 q_2 \dots q_d$ is that of d, we can write:

$$\sqrt{2} = \frac{p_1 p_2 \dots p_n}{q_1 q_2 \dots q_d}. \tag{7.3}$$

It follows that

$$2 = \frac{p_1^2 p_2^2 \dots p_n^2}{q_1^2 q_2^2 \dots q_d^2}. \tag{7.4}$$

As we know from the previous chapter, two different prime numbers p and q squared (or raised to any integer) do not result in two numbers that share factors. The factorisation of p^n is just p^n and that of q^n is just q^n. They are coprime to each other. The fraction in equation 7.4, thus, cannot represent an integer, such as 2. There is only one way for such a fraction to result in an integer, *viz.*, when the numerator is an integer and the denominator is 1, which is obviously not the case for $\sqrt{2}$. It follows that, if the root of an integer is not an integer itself, it is not a rational number either.

But, if $\sqrt{2}$ is not a rational number, a number that can be represented as the fraction of two integers, what the heck is it then?

There are several methods to approximate the number $\sqrt{n}$. The simplest and oldest is the *Babylonian* method, also called *Heron's* method for Heron of Alexandria, a Greek mathematician of the first century who lived in Alexandria.

The idea of Heron's method, basically, is to iteratively approximate the real value starting with a guess. We can start with some arbitrary value. If the first guess, say g, does not equal $\sqrt{n}$, *i.e.* $gg \neq n$, then g is either slightly too big or too small. We either have $gg > n$ or $gg < n$. So, on each step, we improve a bit on the value by taking the average of g and its counterpart n/g. If $gg > n$, then clearly $n/g < g$ and, if $gg < n$, then $n/g > g$. The average of g and a/g is calculated as $(g + a/g)/2$. The result is used as input for the next round. The more iterations of this kind we do, the better is the approximation. In Haskell:

```
heron :: Natural → Natural → Double
heron n s = let a = fromIntegral n in go s a (a / 2)
  where go 0 _ x = x
        go i a x | xx = a     = x
                 | otherwise = go (i − 1) a ((x + a / x) / 2)
```

The function takes two arguments. The first is the number whose square root we want to calculate and the second is the number of iterations we want to do. We then call *go* with s, the number of iterations, a, the *Double* representation of n, and our first guess $a/2$.

In *go*, if we have reached $i = 0$, we yield the result x. Otherwise, we call *go* again with $i - 1$, a and the average of x and a/x.

Let us compute $\sqrt{2}$ following this approach for, say, five iterations. We first have

go 5 2 1 = *go* 4 2 ((1 + 2) / 2)
go 4 2 1.5 = *go* 3 2 ((1.5 + 2 / 1.5) / 2)
go 3 2 1.416666 = *go* 2 2 ((1.416666 + 2 / 1.416666) / 2)
go 2 2 1.414215 = *go* 1 2 ((1.414215 + 2 / 1.414215) / 2)
go 1 2 1.414213 = *go* 0 2 ((1.414213 + 2 / 1.414213) / 2)
go 1 2 1.414213 = 1.414213.

Note that we do not show the complete *Double* value, but only the first six digits. The results of the last two steps, therefore, are identical. They differ, in fact, at the twelfth digit: 1.4142135623746899 (*heron* 2 4) versus 1.414213562373095 (*heron* 2 5). Note that the result of five iterations has one digit less than that of four iterations. This is because that after the last digit, 5, the digit 0 follows and then the *precision* of the Double number type is exhausted. *Irrational* numbers, this is the designation of the type of numbers we are talking about, consist of infinitely many digits. Therefore, the last digit in the number presented above is in fact not the last digit of the number. With slightly higher precision, we would see that the number continues like 03...

The result of (*heron* 2 5) ↑ 2 is fairly close to 2: 1.9999999999999996. It will not get any closer using *Double* representation.

7.2. Φ

Another interesting irrational number is τ or Φ, known as the *divine proportion* or *golden ratio.* The golden ratio is the relation of two quantities, such that the greater relates to the lesser as the sum of both to the greater. This may sound confusing, so here are two symbolic representations:

$$\frac{a}{b} = \frac{a+b}{a}. \tag{7.5}$$

In this equation, a is the greater number and b the lesser. The equation states that the relation of a to b equals the relation of $a + b$ to a. Alternatively we can say

$$\frac{b}{a} = \frac{a}{b-a}. \tag{7.6}$$

Here b is the sum of the two quantities and a is the greater one. The equation states that the relation of b to a is the same as the relation of a to $b - a$, which, of course, is then the smaller quantity.

The golden ratio is known since antiquity. The symbol Φ is an homage to ancient Greek sculptor and painter Phidias (480 – 430 BC). Its first heyday after antiquity was during Renaissance and then it was again extremely popular in the 18^{th} and 19^{th} centuries. Up to our times, artists, writers and mathematicians have repeatedly called the golden ratio especially pleasing.

From equation 7.6 we can derive the numerical value of Φ in the form $\frac{\Phi}{1}$. We can then calculate b, for any value a, as $a\Phi$. The derivation uses some algebraic methods, which we will study more closely in the next part, especially *completing the square*. As such this section is also an algebra teaser.

We start by setting $\Phi = \frac{b}{a}$ with $a = 1$. We then have on the left side of the equation $\Phi = \frac{b}{1} = b$. On the right-hand side, we have $\frac{a=1}{\Phi-1}$:

$$\Phi = \frac{1}{\Phi - 1}. \tag{7.7}$$

Multiplying both sides by $\Phi - 1$, we get

$$\Phi^2 - \Phi = 1. \tag{7.8}$$

This is a quadratic equation and, with some phantasy, we even recognise the fragment of a binomial formula on the left side. A complete binomial formula would be

$$(a - b)^2 = a^2 - 2ab + b^2. \tag{7.9}$$

We try to pattern match the fragment above such that Φ^2 is a^2 and $-\Phi$ is $-2ab$. That would mean that the last term, b^2 would correspond to the square half of the number that is multiplied by Φ to yield $-\Phi$. $-\Phi$ can be seen as $-1 \times \Phi$. Half of that number is $-\frac{1}{2}$. That squared is $\frac{1}{4}$. So, if we add $\frac{1}{4}$ to both sides of the equation, we would end up with a complete binomial formula on the left side:

$$\Phi^2 - \Phi + \frac{1}{4} = 1 + \frac{1}{4} = \frac{5}{4}. \tag{7.10}$$

We can apply the binomial theorem on the left side and get

$$\left(\Phi - \frac{1}{2}\right)^2 = \frac{5}{4}. \tag{7.11}$$

Now we take the square root:

$$\Phi - \frac{1}{2} = \frac{\pm\sqrt{5}}{2}. \tag{7.12}$$

Note that the ± in front of $\sqrt{5}$ reflects the fact that the square root of 5 (or any other number) may be positive or negative. Both solutions are possible. However, since we are looking for a positive relation, we only consider the positive solution and ignore the other one. We can therefore simplify to

$$\Phi - \frac{1}{2} = \frac{\sqrt{5}}{2}. \tag{7.13}$$

Finally, we add $\frac{1}{2}$ to both sides:

$$\Phi = \frac{1+\sqrt{5}}{2} \tag{7.14}$$

and voilá that is Φ. The numerical value is approximately $1.618\,033\,988\,749\,895$. We can define it as a constant in Haskell as

```
phi :: Double
phi = 0.5 * (1 + sqrt 5)
```

We can now define a function that, for any given a, yields b:

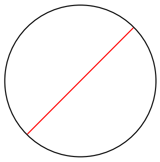
```
golden :: Double → Double
golden a = a * phi
```

golden 1, of course, is just Φ, *i.e.* $1.618\,033\,988\,749\,895$. *golden* 2 is twice that value, namely $3.236\,067\,977\,499\,79$. Furthermore, we can state that 2 / (*golden* 2 − 2), which is $\frac{a}{b-a}$, is again an approximation of Φ.

That is very nice. But there is more to come. Φ, in fact, is intimately connected to the Fibonacci sequence, as we will se in the next chapter.

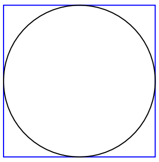
7.3. π

π is probably the most famous irrational number. It emerged in antique mathematics in studying the circle where it expresses the relation between the diameter (depicted in red in the image below) and the circumference (depicted in black):

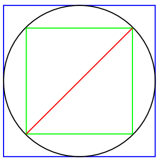


Since, often, the radius, which is half the diameter, is much more important in mathematics, it has been proposed to use $\tau = 2\pi$ where π is used today. But π has survived through history and, even though slightly suboptimal in some situations, it is still in use today.

The reason why the perimeter instead of the radius was used to define the circle constant is probably because classic approaches to approximate π take the perimeter as basis. They start by drawing a square with side length 1 and inscribe a circle into the square with perimeter 1:



Since the square has side length 1, its perimeter, the sum of all its sides is $1+1+1+1 = 4$ and, as we can see clearly in the picture above, this perimeter is greater than that of the circle. 4, hence, is an upper bound for the circumference of the circle with perimeter 1. A lower bound would then be given by a square inscribed in the circle, such that the distance between its opposing corners (red) is 1, the perimeter of the circle:



We see two right triangles with two green sides on a red basis. The basis is the perimeter of the circle, of which we know that its length is 1. You certainly know the Pythagorean theorem, probably the most famous or notorious theorem of all mathematics, which states that, in a right triangle, one with a right angle, an angle of 90°, the sum of the squares of the sides to the left and right of that angle (the green sides) equals the square of the hypothenuse, the red side, which is opposite to the right angle. This can be stated as:

$$a^2 = b^2 + c^2, \tag{7.15}$$

where a is the red side, whose length we know, namely 1. We further know that the green sides are equal. We hence have:

$$1^2 = 2b^2. \tag{7.16}$$

and further derive

$$1 = \sqrt{2b^2}, \tag{7.17}$$

which is

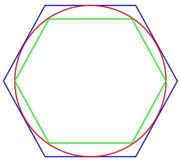
$$1 = \sqrt{2}b. \tag{7.18}$$

Dividing both sides by $\sqrt{2}$, we get

$$b = \frac{1}{\sqrt{2}}, \tag{7.19}$$

the side length of the green square, which is approximately 0.707. The perimeter of the inner square is thus 4×0.707, which is approximately 2.828. Thus π is some value between 2.828 and 4.

That result is not very satifactory, of course. There is room for a lot of numbers between 2.828 and 4. The method was therefore extended by choosing polygons with more than four sides to come closer to the real value of π. The ancient record holder for approximating π is Archimedes who started off with a hexagon, which is easy to construct with compass and ruler:



Then he subsequently doubled the number of sides of the polygon, so that he obtained polygons with 12, 24, 48 and, finally, 96 sides. With this approach he concluded that $\frac{223}{71} < \pi < \frac{22}{7}$, which translates to a number between 3.1408 and 3.1428 and is pretty close to the approximated value 3.14159.

In modern times, mathematicians started to search for approximations by other means than geometry, in particular by infinite series. One of the first series was discovered by Indian mathematician Nilakantha Somayaji (1444 – 1544). It goes like

$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \dots \tag{7.20}$$

We can implement this in Haskell as

```
nilak :: Int → Double
nilak i     | even i      = nilak (i + 1)
            | otherwise = go i 2 3 4
  where go 0 _ _ _ = 3
        go n a b c = let k  | even n = −4
                            | otherwise = 4
                     in (k / (a ∗ b ∗ c)) + go (n − 1) c (c + 1) (c + 2)
```

Here we use a negative term, whenever n, the counter for the step we are performing, is even. Since, with this approach, an even number of steps would produce a bad approximation, we perform, for i even, $i + 1$ and hence an odd number of steps. This way, the series converges to 3.14159 after about 35 steps, *i.e. nilak* 35 is some number that starts with 3.14159.

An even faster convergence is obtained by the beautiful series discovered by French mathematician François Viète (1540 – 1603) in 1593:

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \times \frac{\sqrt{2+\sqrt{2}}}{2} \times \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} \times \dots \tag{7.21}$$

In Haskell this gives rise to a very nice recursive function:

```
vietep :: Int → Double
vietep i = 2 / (go 0 (sqrt 2))
  where go n t | n ≡ i       = 1
               | otherwise = (t / 2) ∗ go (n + 1) (sqrt (2 + t))
```

The approximation 3.14159 is reached with *vietep* 10.

There are many other series, some focusing on early convergence, others on beauty. An exceptionally beautiful series is that of German polymath Gottfried Wilhelm Leibniz (1646 – 1716), who we will get to know more closely later on:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots \tag{7.22}$$

In Haskell this is, for instance:

```
leipi :: Int → Double
leipi i = 4 ∗ go 0 1
  where go n d | n ≡ i       = 0
               | otherwise = let x | even n = 1
                                   | otherwise = −1
                             in x / d + go (n + 1) (d + 2)
```

This series converges really slowly. We reach 3.14159 only after about 400 000 steps.

π appears quite often in mathematics, particularly in geometry. But there are also some unexpected entries of this number. The inevitable Leonhard Euler solved a function, which today is called *Riemann zeta function*, for the special case $s = 2$:

$$\zeta(s) = \frac{1}{1^s} + \frac{1}{2^s} + \frac{1}{3^s} + \dots = \sum_{n=1}^{\infty} \frac{1}{n^s}. \tag{7.23}$$

Euler showed that, for the special case $s = 2$, $\zeta(s)$ converges to $\frac{\pi^2}{6}$; in fact, for any n, n a multiple of 2, $\zeta(n)$ converges to some fraction of a power of π, *e.g.* $\zeta(4)$ approaches $\frac{\pi^4}{90}$, $\zeta(6)$ approaches $\frac{\pi^6}{945}$ and so on.

This is surprising, because the zeta function is not related to circles, but to number theory. It appears for example, when calculating the probability of two numbers being coprime to each other. Two numbers are coprime if they do not share prime factors. The probability of a number being divisible by a given prime p is $\frac{1}{p}$, since every p^{th} number is divisible by p. For two independently chosen numbers, the probability that both are divisible by prime p is therefore $\frac{1}{p} \times \frac{1}{p} = \frac{1}{p^2}$. The reverse probability that both are not divisible by that prime, hence, is $1 - \frac{1}{p^2}$. The probablitiy that there is no prime at all that divides both is then

$$\prod_{p}^{\infty} 1 - \frac{1}{p^2}. \tag{7.24}$$

To cut a long story short, this equation can be transformed into the equation

$$\frac{1}{1 + \frac{1}{2^2} + \frac{1}{3^2} + \dots} = \frac{1}{\zeta(2)} = \frac{1}{\frac{\pi^2}{6}} = \frac{6}{\pi^2} = 0.607 \approx 61\% \tag{7.25}$$

and with this π appears as a constant in number theory expressing the probability of two randomly chosen numbers being coprime to each other.

7.4. *e*

The Bernoullis were a family of Huguenots from Antwerp in the Spanish Netherlands from where they fled the repression by the Catholic Spanish authorities, first to Frankfurt am Main, later to Basel in Switzerland. Among the Bernoullis, there is a remarkable number of famous mathematicians who worked in calculus, probability theory, number theory and many areas of applied mathematics. One of the Basel Bernoullis was Johann Bernoulli (1667 – 1748) who worked mainly in calculus and tutored famous mathematicians like Guillaume L'Hôpital, but whose greatest contribution to the history of math was perhaps to recognise the enormous talent of another of his pupils whose name was Leonhard Euler.

His brother Jacob Bernoulli (1655 – 1705), who worked, as his brother, in calculus, but most prominently in probability theory, is much better known today, partly perhaps because many of Johann's achievements were published under the name of L'Hôpital. Unfortunately, early modern mathematics and science in general was plagued with disputes over priorities in the authorship of contributions, a calamity that authors and authorities later tried to solve by introducing the *droite d'auteur*, better known in the English speaking world as *copyright*.

Among the many problems Jacob studied was the calculation of interests. He started off with a very simple problem. Suppose we have a certain amount of money and a certain interest credited after a given amount of time. To keep it simple, let the amount equal 1 (of any currency of your liking – currencies in Jacob's lifetime were extremely complicated, so we better ignore that detail). After one year 100% interest is paid. After that year, we hence have $1 + \frac{1 \times 100}{100} = 2$ in our account. That is trivial. But what, if the interest is paid in shorter periods during the year? For instance, if the interest is paid twice a year, then the interest for that period would be 50%. After six months we would have $1 + \frac{1 \times 50}{100} = 1.5$ in our account. After one year, the account would then be $1.5 + \frac{1.5 \times 50}{100} = 1.5 + \frac{75}{100} = 1.5 + 0.75 = 2.25$.

Another way to see this is that the initial value is multiplied by 1.5 (the initial value plus the interest) twice: $1 \times 1.5 \times 1.5 = 1 \times 1.5^2 = 2.25$. When we reduce the period even further, say, to three months, then we had $1.25^4 \approx 2.4414$. On a monthly base, we would get $\left(1 + \frac{1}{12}\right)^{12} \approx 2.613$. On a daily basis, we would have $\left(1 + \frac{1}{365}\right)^{365} \approx 2.7145$. With hourly interests and the assumption that one year has $24 \times 365 = 8760$ hours, we would get $\left(1 + \frac{1}{8760}\right)^{8760} \approx 2.71812$. With interest paid per minute we would get $\left(1 + \frac{1}{525600}\right)^{525600} \approx 2.71827$ and on interest paid per second, we would get $\left(1 + \frac{1}{3156000}\right)^{3156000} \approx 2.71828$. In general, for interest on period n, we get:

$$\left(1 + \frac{1}{n}\right)^n.$$

You may have noticed in the examples above that this formula converges with greater and greater ns. For n approaching ∞, it converges to 2.71828, a number that is so beautiful that we should look at more than just the first 5 digits:

$$2.7\ 1828\ 1828\ 4590\ 4523\ldots$$

This is e. It is called Euler's number or, for the first written appearance of concepts related to it in 1618, Napier's number. It is a pity that its first mentioning was not in the year 1828. But who knows – perhaps in some rare Maya calendar the year 1618 actually is the year 1828.

An alternative way to approach e that converges much faster than the closed form above is the following:

$$1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \dots$$

or, in other words:

$$e = \sum_{n=1}^{\infty} \frac{1}{n!}. \tag{7.26}$$

We can implement this equation in Haskell as

```
e_ :: Integer → Double
e_ p = 1 + sum [1 / (dfac n) | n ← [1 . . p]]
  where dfac = fromInteger ∘ fac
```

After some experiments with this function, we see that it converges already after 17 recursions to a value that does not change with greater arguments at *Double* precision, such that `e_ 17 ≡ e_ 18 ≡ e_ 19 ≡` ... We could then implement e as

```
e :: Double
e = e_ 17
```

The fact that e is related to the factorial may lead to the suspicion that it also appears directly in a formula dealing with factorials. There, indeed, is a formula derived by James Stirling who we already know for the Stirling numbers. This formula approximates the value of $n!$ without the need to go through all the steps of its recursive definition. Stirling's formula is as follows:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{7.27}$$

This equation is nice already because of the fact that e and π appear together to compute the result of an important function. But how precise is the approximation? To answer this question, we first implement Stirling's formula:

```
stirfac :: Integer → Integer
stirfac i = ceiling $ (sqrt (2 ∗ pi ∗ n)) ∗ (n / e) ↑ i
  where n = fromIntegral i
```

Note that we *ceil* the value, instead of rounding it just to the next integer value.

Then we define a function to compute the difference *difac n* = *fac n* − *stirfac n*. The result for the first 15 numbers is

$$0, 0, 0, 0, 1, 9, 59, 417, 3343, 30104, 301174, 3314113, 39781324, 517289459, 7243645800.$$

For the first numbers, the difference is 0. Indeed:

$$\begin{array}{ccccr} 1! & = & stirfac(1) & = & 1 \\ 2! & = & stirfac(2) & = & 2 \\ 3! & = & stirfac(3) & = & 6 \\ 4! & = & stirfac(4) & = & 24 \end{array}$$

Then, the functions start to disagree, for instance $5! = 120 \neq stirfac(5) = 119$. The difference grows rapidly and reaches more than 3 million with 12!. But what is the deviation in relation to the real value? We define the function 100 ∗ (*fromIntegral* $ *difac n*) / (*fromIntegral* $ *fac n*) to obtain the difference in terms of a percentage of the real value. We see starting from 5 (where the first difference occurs):

$$0.8333, 1.25, 1.1706, 1.0342, 0.9212, 0.8295, 0.7545, 0.6918, 0.6388, 0.5933, 0.5539, \ldots$$

For 5, the value jumps up from 0 to 0.8333%, climbs even higher to 1.25% and then starts to descrease slowly. At 42 the deviation falls below 0.2%. At 84, it falls below 0.1% and keeps falling. Even though the difference appears big in terms of absolute numbers, the percentage quickly shrinks and, for some problems, may even be neglible.

A completely different way to approximate e is by *continued fractions*. Continued fractions are infinite fractions, where each denominator is again a fraction. For instance:

$$e = 1 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \frac{1}{\ldots}}}}}} \tag{7.28}$$

A more readable representation of continued fractions is by sequences of the denominator like:

$$e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, \ldots] \tag{7.29}$$

where the first number is separated by a semicolon to highlight the fact that it is not a denominator, but an integral number added to the fraction that follows. We can capture this very nicely in Haskell, using just a list of integers. However, in some cases we might have a fraction with numerators other than 1. An elegant way to represent this case is by using fractions instead of integers. We would then represent $\frac{2}{a+\dots}$ as $\frac{1}{\frac{1}{2}a+\dots}$. Here is an implementation:

```
contfrac :: [Quoz] → Double
contfrac [] = 1
contfrac [i] = fromQuoz i
contfrac (i : is) = n + 1 / (contfrac is)
  where n = fromQuoz i

fromQuoz :: Quoz → Double
fromQuoz i = case i of
  (Pos (Q nu d)) → fromIntegral nu / fromIntegral d
  (Neg (Q nu d)) → negate (fromIntegral nu / fromIntegral d)
```

For *contfrac* $[2,1,2,1,1,4,1,1,6]$ we get 2.7183, which is not bad, but not yet too close to e. With $[2,1,2,1,1,4,1,1,6,1,1,8,1,1,10]$ we get 2.718 281 828, which is pretty close.

Examining the sequence a bit further, we see that is has a regular structure. We can generate it by means of the *Engel expansion* named for Friedrich Engel (1861 – 1941), a German mathematician who worked close with the great Norwegian algebraist Sophus Lie (1842 – 1899). The Engel expansion can be implemented as follows:

```
engelexp :: [Integer]
engelexp = 2 : 1 : go 1
  where go n = (2 * n) : 1 : 1 : go (n + 1)
```

The following fraction, however, conoverges must faster than the Engel expansion: $[1, \frac{1}{2}, 12, 5, 28, 9, 44, 13]$. Note that we take advantage of the datatype *Quoz* to represent a numerator that is not 1. This sequence can be generated by means of

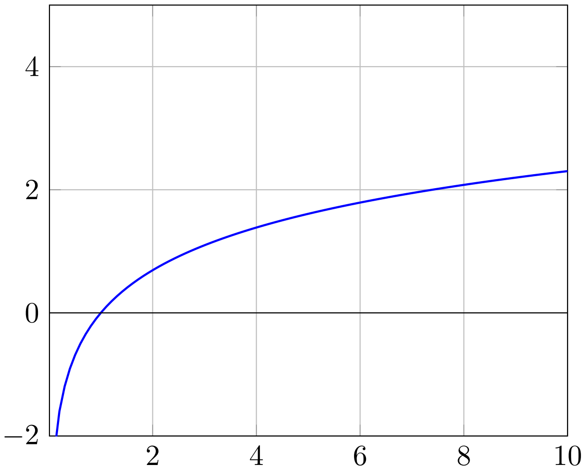
```
fastexp :: [Quoz]
fastexp = 1 : (Pos (1 % 2)) : go 1
  where go n = (16 * n − 4) : (4 * n + 1) : go (n + 1)
```

This fraction converges already after 7 steps to the value 2.718 281 828:

contfrac (*take* 7 *fastexp*).

The area of mathematics where e is really at home is analysis and its vast areas of application, which we will study in the third part of this series. The reason for the prominence of e in analysis stems from the *natural logarithm*, which we already introduced in the first chapter. The natural logarithm of a number n, usually denoted $\ln(n)$, is the exponent x, such that $e^x = n$.

The natural logarithm can be graphed as follows:



The curious fact that earned the natural logarithm its name is that, at $x = 1$, the curve has the slope 1. This might sound strange for the moment. We will investigate that later.

7.5. γ

The *harmonic series* is defined as

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots \tag{7.30}$$

A harmonoic series with respect to a given number k, called the *harmonic number* H_k, then is

$$\sum_{n=1}^{k} \frac{1}{n} = 1 + \frac{1}{2} + \dots + \frac{1}{k} \tag{7.31}$$

This is easily implemented in Haskell as

```
harmonic :: Natural → Double
harmonic n = sum [1 / d | d ← map fromIntegral [1 .. n]]
```

Some harmonic numbers are (*map harmonic* [1 . . 10]):

$1, 1.5, 1.8\overline{3}, 2.08\overline{3}, 2.28\overline{3}, 2.44\overline{9}, 2.5928, 2.7178, 2.8289, 2.9289.$

The harmonic series is an interesting object of study in its own right. Here, however, we are interested in something else. Namely, the difference of the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \to \infty} H_n - \ln(n). \tag{7.32}$$

We can implement this equation as

```
harmonatural :: Natural → Double
harmonatural n = harmonic n − ln n
  where ln = log ∘ fromIntegral
```

Applied on the first numbers with *map harmonatural* $[1\,..\,10]$, the function does not show interesting results:

$1.0, 0.8068, 0.7347, 0.697, 0.6738, 0.6582, 0.6469, 0.6384, 0.6317, 0.6263, \dots$

Applied to greater numbers, however, the results approach a constant value:

harmonatural $100 = 0.58220$
harmonatural $1000 = 0.57771$
harmonatural $10000 = 0.57726$
harmonatural $100000 = 0.57722$

With even greater numbers, the difference converges to $0.577\,21$. This number, γ, was first mentioned by – surprise – Leonhard Euler and some years later by Italian mathematician Lorenzo Mascheroni (1750 – 1800) and is therefore called the Euler-Mascheroni constant.

This mysterious number appears in different contexts and, apparently, quite often as a difference or average. An ingenious investigation was carried out by Belgian mathematician Charles Jean de la Vallée-Poussin (1866 – 1962) who is famous for his proof of the Prime number theorem. Vallée-Poussin studied the quotients of a number n and the primes up to that number. If n is not prime itself, then there are some prime numbers p, namely those of the prime factorisation of n, such that $\frac{n}{p}$ is an integer. For others, this quotient is a rational number, which falls short of the next natural number. For instance, there are four prime numbers less than 10: 2, 3, 5 and 7. The quotients are

$5, 3.\overline{3}, 2, 1.\overline{428571}.$

5 and 2, the quotients of 2 and 5, respectively, are integers and there, hence, is no difference. The quotient $\frac{10}{3} = 3.\overline{3}$, however, falls short of 4 by $0.\overline{6}$ and $\frac{10}{7} = 1.\overline{428571}$ falls short of 2 by $0.\overline{57142828}$.

Vallée-Poussin asked what the average of this difference is. For the example 10, the average is about $0.3\overline{095238}$. One might think that this average, computed for many numbers or for very big numbers, is about 0.5, so that the probability for the quotient of n and a random prime number to fall into the first or the second half of the rational numbers between two integers is equal, *i.e.* 50% for both cases. It turns out it is not. For huge numbers, de la Vallée-Poussin's average converges to 0.577 21, the Euler-Mascheroni constant.

It converges quite slow, however. If we implement the prime quotient as

```
pquoz :: Natural → [Double]
pquoz n = [d / p | p ← ps]
  where ps = map fromIntegral (takeWhile (<n) allprimes)
        d  = fromIntegral n
```

and its average as

```
pquozavg :: Integer → Double
pquozavg n = (sum ds) / (fromIntegral $ length ds)
  where qs = pquoz n
        ns = map (fromIntegral ∘ ceiling) qs
        ds = [n − q | (n, q) ← zip ns qs]
```

we can experiment with some numbers like

pquozavg 10 $=0.3\overline{095238}$
pquozavg 100 = 0.548731
pquozavg 1000 = 0.5590468
pquozavg 10000 = 0.5666399
pquozavg 100000 = 0.5695143
...

With greater and greater numbers, this value approaches γ. Restricting n to prime numbers produces good approximations of γ much earlier. From 7 on, *pquozavg* with primes results in numbers of the form 0.5... *pquozavg* 43 = 0.57416 is already very close to γ. It may be mentioned that 43 is suspiciously close to 42.

With de la Vallée-Poussin's result in mind, it is not too surprising that γ is related to divisors and Euler's totient number. A result of Gauss' immediate successor in Göttingen, Peter Gustav Lejeune-Dirichlet (1805 – 1859), is related to the average number of divisors of the numbers $1 \ldots n$. We have already defined a function to generate the divisors of a number n, namely *divs*. Now we map this function on all numbers up to n:

```
divsupn :: Natural → [[Natural]]
divsupn n = map divs [1 . . n]
```

Applied to 10, this function yields:

$[[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6], [1, 7], [1, 2, 4, 8], [1, 3, 9], [1, 2, 5, 10]]$

For modelling Lejeune-Dirichlet's result, we further need to count the numbers of divisors of each number:

```
ndivs :: Integer → [Int]
ndivs = map length ∘ divsupn
```

Applied again to 10, *ndivs* produces:

$[1, 2, 2, 3, 2, 4, 2, 4, 3, 4]$

Now we compute the average of this list using

```
dirichlet :: Integer -> Double
dirichlet n   = s / l
  where ds = ndivs n
        l  = fromIntegral $ length ds
        s  = fromIntegral $ sum ds
```

For *dirichlet* 10 we see 2.7. This does not appear too spectacular. Greater numbers show:

dirichlet $100 = 4.759$
dirichlet $250 = 5.684$
dirichlet $500 = 6.38$
dirichelt $1000 = 7.069$

As we can see, the number is slowly increasing resembling a log function or, more specifically, the natural log. When we compare the natural log, we indeed see that the results are close:

$\ln 100 = 4.605$
$\ln 250 = 5.521$
$\ln 500 = 6.214$
$\ln 1000 = 6.907$

For greater and greater numbers, the difference of the *dirichlet* function and the natural logarithm approaches

$$0.154435 \approx 2\gamma - 1. \tag{7.33}$$

For the five examples above, the difference is still significantly away from that number:

$\Delta 100 = 0.2148$
$\Delta 250 = 0.1625$
$\Delta 500 = 0.1635$
$\Delta 1000 = 0.1612,$

but already $\Delta 2000 = 0.158$ comes close and $\Delta 4000 = 0.1572$ approaches the value even further.

An important constant derived from γ is e^{γ}, which is a limit often seen in number theory. One instance is the lower bound of the totient function. There is a clear upper bound, namely $n - 1$. Indeed, $\varphi(n)$ can never yield a value greater $n - 1$ and this upper bound is reached exclusively by prime numbers. There is no such linear lower bound. That is,

$\varphi(n)$ can assume values that are much smaller than the value seen for $n-1$ or other numbers less than n. But there is a lower bound that slowly grows with n. This lower bound is often given as $\frac{n}{\ln \ln n}$. This lower bound, however, is too big. There are some values that are still below that border. $\varphi(40)$, for instance, is 16. $\frac{40}{\ln \ln 40}$, however, is around 30. A better, even still not perfect approximation, is

$$\frac{n}{e^{\gamma} \ln \ln n}.$$

For $n = 40$ again, $\frac{40}{e^{\gamma} \ln \ln 40}$ is around 17 and, hence, very close to the real value.

We see that γ is really a quite mysterious number that appears in different contexts, sometimes in quite a subtle manner. The greatest mystery, however, is that it is not so clear that this number belongs here in the first place. Indeed, it has not yet been shown that γ is irrational. In the approximations, we have studied in this section, we actually have not seen the typical techniques to create irrational numbers like roots, continuous fractions and infinite series. If γ is indeed rational, then it must be the fraction of two really large numbers. In 2003, it has been shown that the denominator of such a fraction must be greater than 10^{242080}. A number big enough, for my taste, to speak of *irrational.*

7.6. Representation of Real Numbers

The set of real numbers $\mathbb{R}$ is the union of the rational and the irrational numbers. When we write real numbers on paper, we use the decimal notation. A number in decimal notation corresponds to an ordinary integer terminated by a dot called the decimal point; this integer corresponds to the part of the real number greater 1 or 0 of course. After the dot a stream of digits follows, which is not necessarily a number in the common sense, since it may start with zeros, *e.g.* 0.0001. In fact, one could say that the part after the dot corresponds to a a reversed integer, since the zero following this number have no impact on the value of the whole expression, *i.e.* $0.10 = 0.1$.

Any rational number can be expressed in this system. An integer corresponds just to the part before the dot: $1.0 = 1$. A fraction like $\frac{1}{2}$ is written as 0.5. We will later look at how this is computed concretely. Rationals in decimal notation can be easily identified: all numbers in decimal notation with a finite part after the dot are rational: 0.25 is $\frac{1}{4}$, 0.75 is $\frac{3}{4}$, 0.2 is $\frac{1}{5}$ and so on.

There are some rational numbers that are infinite. For example, $\frac{1}{3}$ is $0.333333\dots$, which we encode as $0.\overline{3}$. Such periodic decimals are easy to convert to fractions. We just have to multiply them by a power of 10, such that there is a part greater 0 before the decimal point and that the first number of the repeating period is aligned to it. For $0.\overline{3}$, this is just $10 \times 0.\overline{3} = 3.\overline{3}$. For $0.1\overline{6}$, it would be $10 \times 0.1\overline{6} = 1.\overline{6}$. For $0.\overline{09}$, it would be $10^2 \times 0.\overline{09} = 9.\overline{09}$. We then subtract the original number from the result. If the original number is x, we now have $10^n x - x = (10^n - 1)x$. For $x = 0.\overline{3}$ this is $9x$; for $x = 0.1\overline{6}$ this, too, is $9x$ and for $x = 0.\overline{09}$ this is $99x$. The results are 3, 1.5 and 9 respectively. We now build a fraction of this result as numerator and the factor (9 or 99) in the denominator. Hence, $0.\overline{3} = \frac{3}{9} = \frac{1}{3}$, $0.1\overline{6} = \frac{1.5}{9} = \frac{3}{18} = \frac{1}{6}$ and $0.\overline{09} = \frac{9}{99} = \frac{1}{11}$.

A curiosity resulting from this calculations is that $9 \times 1/3 = 3$ and $9 \times 0.\overline{3} = 2.\overline{9}$ are actually the same! A correct implementation must take this into account. Try it with Haskell, you will see that $9 * 0.3333333333333333$ is indeed 3. It will not work if you forget a three. The precision of the *Double* number type is 16 digits after the decimal point. With only 15 threes, the result will be 2.9999999999999997.

Irrational numbers in the decimal notation have infinite many digits after the decimal point. With this said, it is obvious that we cannot represent irrational numbers in this system. We can of course represent any number by some kind of formula like $\sqrt{5}$ and do some math in this way such as $\frac{1+\sqrt{5}}{2}$, *etc.* But often, when we are dealing with applied mathematics, such formulas are not very useful. We need an explicit number. But, unfortunately or not, we have only limited resources in paper, brainpower and time. That is, at some point we have to abandon the calculations and work with what can be achieved with the limited resources we have at our disposal.

The point in time at which we take the decision that we now have calculated enough is the measure for the precision of the real number type in question. On paper, we would hence say that we write only a limited number of digits after the decimal point. In most day-to-day situations where real numbers play a role, like in dealing with money, cooking, medication or travelling distances, we calculate up to one or two decimal places. Prices, for instances are often given as 4.99 or something, but hardly as 4.998. Recipes would tell that we need 2.5 pounds or whatever of something, using one decimal place. One would say that it is about 1.5km to somewhere, but hardly that it is 1.499961km. In other areas, especially in science much more precision is needed. We therefore need a flexible datatype.

A nice and clean format to represent real numbers uses two integers or, as the following definition, two natural numbers:

data *RealN* = *R Natural Natural*

The first number represents the integral part. You will remember that a number is a list of digits where every digit is multiplied by a power of ten according to the place of the digit. The first digit, counted from the right, is multiplied by $10^0 = 1$. The second is multiplied by 10^1, the third by 10^2 and so on.

The digit multiplied by 10^0, is the last digit before the decimal point. If we wanted to push it to the right of the decimal point, we would need to reduce the exponent. So, we would multiply it not by 10^0, but by 10^{-1} to push it to the first decimal place. This is the function of the second number in the datatype above. It represents the value of the least significant bit in terms of the exponent to which we have to raise 10 to obtain the number represented by this datatype. Since our datatype uses a natural number, we have to negate it to find the exponent we need.

For instance, the number $R\ 25\ 2$ corresponds to 25×10^{-2}, which we can reduce stepwise to 2.5×10^{-1} and 0.25×10^{0}. A meaningful way to show this datatype would therefore be:

```
instance Show RealN where
  show (R a e) = show a ++ "*10^(-" ++ show e ++ ")"
```

There are obviously many ways to represent the same number with this number type. 1, for instance, can be represented as

```
one = R 1 0
one = R 10 1
one = R 100 2
one = R 1000 3
...
```

To keep numbers as concise as possible, we define a function to simplify numbers with redundant zeros:

```
simplify :: RealN → RealN
simplify (R 0 _) = R 0 0
simplify (R a e) | e > 0 ∧
                   a `rem` 10 ≡ 0 = simplify (R (a `div` 10) (e − 1))
                 | otherwise      = R a e
```

As long as the exponent is greater 0 and the base *a* is divisible by 10, we reduce the exponent by one and divide *a* by 10. In other words, we remove unnecessary zeros. The following constructor uses *simplify* to create clean real numbers:

```
real :: Natural → Natural → RealN
real i e = simplify (R i e)
```

7.7. ℝ

We now define how to check two real numbers for equality:

```
instance Eq RealN where
  r1@(R a e1) ≡ r2@(R b e2) | e1 ≡ e2 = a ≡ b
                            | e1 > e2 = r1 ≡ blowup e1 r2
                            | e1 < e2 = blowup e2 r1 ≡ r2
```

If the exponents are equal, then we trivially compare the coefficients. Otherwise, we first expand the number with the smaller exponent using *blowup*:

```
blowup :: Natural → RealN → RealN
blowup i (R r e) | i ⩽ e      = R r e
                 | otherwise = R (r ∗ 10 ↑ (i − e)) i
```

That is simple! If the target i is greater than the current exponent of the number, we just multiply the coefficient by 10 raised to the difference of the target exponent and the current exponent and make the target the new exponent. Otherwise, nothing changes.

We continue with comparison, which follows exactly the same logic:

```
instance Ord RealN where
  compare r1@(R a e1) r2@(R b e2) | e1 ≡ e2 = compare a b
                                  | e1 > e2 = compare r1 (blowup e1 r2)
                                  | e1 < e2 = compare (blowup e2 r1) r2
```

Now we make *RealN* instance of *Num*:

```
instance Num RealN where
  (R a e1) + (R b e2) | e1 ≡ e2   = simplify $ R (a + b) e1
                      | e1 > e2   = simplify $ R (a + b ∗ 10 ↑ (e1 − e2)) e1
                      | otherwise = simplify $ R (a ∗ 10 ↑ (e2 − e1) + b) e2
  (R a e1) − (R b e2) | e1 ≡ e2 ∧
                        a ⩾ b     = simplify $ R (a − b) e1
                      | e1 ≡ e2   = error "subtraction beyond zero!"
                      | e1 > e2   = simplify $ (R a e1) − (R (b ∗ 10 ↑ (e1 − e2)) e1)
                      | otherwise = simplify $ (R (a ∗ 10 ↑ (e2 − e1)) e2) − (R b e2)
  (R a e1) ∗ (R b e2) = real (a ∗ b) (e1 + e2)
  negate  r     = r  -- we cannot negate natural numbers
  abs     r     = r
  signum  r     = r
  fromInteger i = R (fromIntegral i) 0
```

Addition is again the same logic. For two numbers with equal exponents, we just add the coefficients. If the exponents differ, we first convert the smaller number to the greater exponent.

For subtraction, note that we define *RealN* like numbers before without negatives. To consider signedness, we still have to use the datatype *Signed RealN*. Consequently, we have to rule out the case where the first number is smaller than the second one.

Multiplication is interesting. We multiply two real numbers by multiplying the coefficients and adding the exponents. We have already seen this logic, when defining the natural number type. Some simple examples may convince you that this is the right way to go. 1×0.1, for instance, is 0.1. In terms of our *RealN* type, this corresponds to $(R\ 1\ 0) * (R\ 1\ 1) \equiv (R\ (1 * 1)\ (0 + 1)$.

The next task is to make *RealN* instance of *Fractional*:

```
instance Fractional RealN where
  (/) = rdiv 17
  fromRational r = (R (fromIntegral $ R.numerator r) 0) /
                   (R (fromIntegral $ R.denominator r) 0)
```

The method *fromRational* is quite simple. We just create two real numbers, the numerator of the original fraction and its denominator, and the we divide them. What we need to do this, of course, is division. Division, as usual, is a bit more complicated than the other arithmetic operations. We define it as follows:

```
rdiv :: Natural → RealN → RealN → RealN
rdiv n r1@(R a e1) r2@(R b e2) | e1 < e2 =
                                   rdiv n (blowup e2 r1) r2
                                 | a < b ∧ e1 ≡ e2 =
                                   rdiv n (blowup (e2 + 1) r1) r2
                                 | otherwise =
                                   simplify (R (go n a b) (e1 − e2 + n))
  where go i x y | i ⩽ 0 = 0
                 | otherwise =
                   case x `quotRem` y of
                     (q, 0) → 10 ↑ i ∗ q
                     (q, r) → let (r′, e)    = borrow r y
                                  q′         = 10 ↑ i ∗ q
                              in if e > i then q′
                                          else q′ + go (i − e) r′ y
```

rdiv has one more argument than the arithmetic operations seen before. This additional argument, n, defines the precision of the result. This is necessary, because, as we will see, the number of iterations the function has to perform depends on the precision the result is expected to have.

If the first number is smaller than the second, either because its exponent or its coefficient is smaller, we blow it up so that it is at least the same size. Then we calculate the new coefficient by means of *go* and the new exponent as the difference of the first and the second exponent plus the expected precision. Note that division has the inverse effect on the size of the exponents as multiplication. When we look again at the example 1 and 0.1, we have $1/0.1 = 10$, which translates to $(R\ 1\ 0)\ /\ (R\ 1\ 1) = R\ (1\ /\ 1)\ (0 - 1)$, which of course is the same as $R\ 10\ 0$.

The inner function *go* proceeds until i, which initially is n, becomes 0 or smaller. In each step, we divide x, initially the coefficient of the first number, and y, the coefficient of the second number. If the result leaves no remainder, we are done. We just raise q to the power of the step in question. Otherwise, we continue dividing the remainder r by y. But before we continue, we borrow from y, that is, we increase r until it is at least y. i is then decremented by the number of zeros we borrowed this way. If we run out of i, so to speak, that is if $e > i$, then we terminate with q raised to the current step.

borrow is just the same as *blowup* applied to two natural numbers:

```
borrow :: Natural → Natural → (Natural, Natural)
borrow a b | a ⩾ b       = (a, 0)
           | otherwise = let (x, e) = borrow (10 ∗ a) b in (x, e + 1)
```

We now make *RealN* instance of *Real*. We need to define just one method, namely how to convert *RealN* to *Rational*, which we do just by creating a fraction with the coefficient of the real number in the numerator and 10 raised to the exponent in the denominator. The rest is done by the *Rational* number type:

```
instance Real RealN where
  toRational (R r e) = i % (10 ↑ x)
    where i  = fromIntegral r :: Integer
          x = fromIntegral e :: Integer
```

We further add a function to convert real numbers to our *Ratio* type:

```
r2R :: RealN → Ratio
r2R (R a e) = ratio a (10 ↑ e)
```

We also add a function to convert our real number type to the standard *Double* type:

```
r2d :: RealN → Double
r2d r@(R a e) | e > 16      = r2d (roundr 16 r)
              | otherwise = (fromIntegral a) / 10 ↑ e
```

An inconvenience is that we have to round a number given in our number type so it fits into a *Double*. For this, we assume that the *Double* has a precision of 16 decimal digits. This is not quite true. The *Double* type has room for 16 digits. But if the first digits after the decimal point are zeros, the *Double* type will present this as a number raised to a negative exponent, just as we do with our real type. In this case, the *Double* type may have a much higher precision than 16. For our purpose, however, this is not too relevant.

So, here is how we round:

```
roundr :: Natural → RealN → RealN
roundr n (R a e) | n ⩾ e       = R a e
                 | otherwise = let b  = a ‘div‘ 10
                                   l  = a − 10 ∗ b
                                   d | l < 5 = 0
                                     | otherwise = 1
                               in roundr n (R (b + d) (e − 1))
```

That is, we first get the least significant digit l as $l = a - 10 * (div\ a\ 10)$, where, if *div* is the Euclidian division, the last digit of a remains. If the last digit is less than 5, we just drop it off. Otherwise, we add 1 to the whole number. If we now define

$one = R\ 1\ 0$
$three = R\ 3\ 0$
$third = one\ /\ three,$

then *roundr* 16 (*three* $*$ *third*) yields 1, as desired.

Finally, we define

type *SReal* = *Signed RealN*

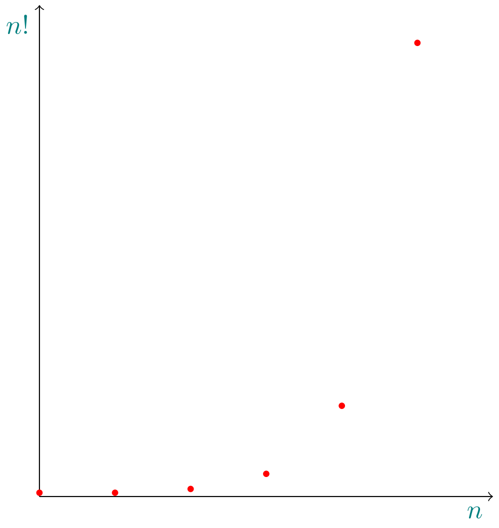
and have a fulfledged real number datatype.

7.8. Real Factorials

As we have already done in the domain of integers, we will now try to generalise some of the combinatorial sequences to the domain of real numbers. We did this with integers for the binomial coefficients. Before we can introduce real binomials, however, we need to look at factorials. The factorial of n, $n!$, is the number of possible permutations of a sequence of n elements. This is a very concrete and easy to grasp concept. A function, however, that results in a fraction or even an irrational number does not count anything similar to that. It may measure a *continuous* quantity (a weight or distance, for example), but certainly not a *discrete* value like a counting result.

This leads to a didactical dilemma that often arises in modern mathematics. Mathematics usually aims to generalise concepts and often independent of concrete applications of this generalised concept. The applications often follow, sometimes years or even centuries later. We all know geometry in the two-dimensional plane. Mathematicians have generalised the concepts of two-dimensional geometry to n dimensions. From a naïve perspective looking only at immediate applicability, there is not much sense in such geometries beyond three or, with Einstein in mind, four dimensions. However, the areas of mathematics that study *space* with more than four or even infinitely many dimensions (such as linear algebra and complex analysis), actually have many applications in statistics, engineering, physics and other rather practical domains. Furthermore, applications is an important, but not the only motivation for mathematical investigation. Mathematics studies its fundamental concepts (like numbers, sets or the space in which we exercise geometry) to arrive at general theorems. So, even when there is no immediate application yet, for the sake of better understanding of the concepts involved, mathematicians do not hesitate to ask questions that appear to be absurd or meaningless to "ordinary" people (whoever those are).

An obvious way to look at real factorials is to represent them in the Cartesian plane, *i.e.* in the coordinate system. We can sketch the factorials of natural numbers as:



where the factorials of n are shown on the vertical axis (the y-axis) with a scale of $1:10$ in relation to the values for n on the horizontal axis (the x-axis). The diagram shows the factorials of the numbers $n \in \{0,1,2,3,4,5\}$, which are $1,1,2,6,24,120$. The factorials are drawn as dots with the coordinates $(n, n!)$. The space between these dots is empty. A continuous interpretation of the factorials would ask for the values within this white space and aim to find a curve that connects the discrete dots in the picture.

One obvious requirement for such a function f is that for any $n \in \mathbb{N}, f(n) = n!$. A function that (almost) fulfils this requirement is the *Gamma function*, Γ. For any integer $n > 1$, Γ is

$$\Gamma(n) = (n-1)\Gamma(n-1) \tag{7.34}$$

with $\Gamma(1) = 1$. We can model this in Haskell as

```
gamman :: Natural → Natural
gamman 0 = ⊥
gamman 1 = 1
gamman n = (n − 1) ∗ gamman (n − 1)
```

When we apply this to the numbers $1 \dots 10$, *map gamman* $[1\,..\,10]$, we see:

$[1,1,2,6,24,120,720,5040,40320,362880]$

There is a snag. Here are the factorials, created with *map fac* $[1\,..\,10]$:

$[1,2,6,24,120,720,5040,40320,362880,3628800]$

The Γ function, hence, creates the factorials, but shifted down by one. Indeed, we have

$$\Gamma(n+1) = n! \tag{7.35}$$

With $[\mathit{gamman}\ (n+1) \mid n \leftarrow [1\mathinner{..}10]]$, we finally see the factorials in their correct places in the sequence:

$[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]$

That the Γ function is defined like this, is for historical reasons and does not need to bother us here. We only have to keep in mind that, whenever we want to make the connection from Γ to factorial, we need to increment n by 1.

Well, the *gamman* function above shows us just another way to express factorials for natural numbers. But we wanted to find a function for real numbers. There are indeed many ways to define the Γ function in that domain. The canonical way is the *Euler integral of the second kind*, but, since we have not yet introduced integrals, we choose another way that we already know, namely infinite products.

We first look at the following product, which was found already by Euler:

$$\Gamma(x) = \frac{1}{x}\prod_{n=1}^{\infty}\frac{(1+\frac{1}{n})^x}{1+\frac{x}{n}} \tag{7.36}$$

We can reformulate this equation in Haskell as:

```
gammal :: Natural → RealN → RealN
gammal i x = (1 / x) ∗ product [(1 + 1 / n) ∗∗ x / (1 + x / n) | n ← [1 .. m]]
  where m = fromIntegral i
```

The function receives a *Natural* and a *RealN*. The *Natural*, i, is just the number of iterations we want to perform, since we do not have the time to go through all iterations of the infinite product. The *go* function implements the product itself. Finally we multiply $\frac{1}{x}$ and we are done.

Let us look at how precise this function can mimic $n!$ for a given number of iterations. When we apply *gammal* on $x = 1$, then we obviously get 1, since we compute

$$\frac{1}{1}\prod_{n=1}^{\infty}\frac{(1+\frac{1}{n})^1}{1+\frac{1}{n}} = \prod_{n=1}^{\infty}\frac{1+\frac{1}{n}}{1+\frac{1}{n}} = 1 \times 1 \times 1 \times \cdots = 1.$$

Indeed, *gammal* 1 1 immediately yields 1. Next, we try *gammal* 1 2 and get

$$0.\overline{6}$$

That is far off the expected value 1. We increase the number of iterations and try *gammal* 10 2:

$$0.91666666\dots$$

Already better, but still not 1. We try *gammal* 100 2 and see:

$$0.99019607\dots$$

and *gammal* 1000 2:

$$0.99900199\dots$$

We see, the function converges slowly. When we try the factorials for $3\dots6$ with 1000 iterations, we see:

$$1.99401993\dots, 5.96418512\dots, 23.76178831\dots, 118.21843985\dots$$

The first two values are fairly close to the expected results. $\Gamma(5)$ and $\Gamma(6)$, which should be 24 and 120, however, are clearly off. We try *gammal* 10000 5 and see:

$$23.97601798\dots$$

Not good, but much better. What about *gammal* 10000 6? Here it is:

$$119.82018583\dots$$

Still more than 0.1 off the expected result 120. We try again with 100 000 iterations:

$$119.98200185\dots$$

That was still not enough! Let us try with one million iterations:

$$119.99820001\dots$$

We see that the infinite product slowly approaches the expected results of the Γ function, but the stress here is on "slowly". Already for $x = 6$, we need a lot of iterations to achieve a deviation of less than 0.01.

Let us look at another infinite product. It is not faster than the one we looked at – on the contrary, it is even slower – but it is a nice formula:

$$\Gamma(x) = \frac{e^{-\gamma x}}{x} \prod_{n=1}^{\infty} \left(1 + \frac{x}{n}\right)^{-1} e^{\frac{x}{n}}, \tag{7.37}$$

where γ is the Euler-Mascheroni constant and e, the Euler-Napier constant. This formula was found by German mathematician Karl Weierstrass (1815 – 1897) who was instrumental in the foundations of modern analysis. In Haskell, his definition of the Γ function may look like:

```
gammae :: Natural → RealN → RealN
gammae i x = f ∗ product [e ∗∗ (x / n) ∗ (1 + (x / n)) ∗∗ (−1) | n ← [1 . . m]]
  where f = e ∗∗ ((−1) ∗ gamma ∗ x) / x
        m = fromIntegral (i − 1)
```

An important difference to the first formula is that, from this one, it is not obvious that it should result in 1 for $x = 1$. Let us give it a try. *gammae* 1 1:

$$0.56145836\ldots$$

That is an ugly result! Already for 1, it diverges from the expected result by almost $\frac{1}{2}$! Let us see how many iterations we need to approach 1:

gammae 10 1 = 0.95043595...
gammae 100 1 = 0.99500219...
gammae 1000 1 = 0.99949804...

and so on. What about 2?

gammae 1 2 = 0.15761774...
gammae 10 2 = 0.82120772...
gammae 100 2 = 0.98022710...
gammae 1000 2 = 0.99799833...

It definitely converges slower than Euler's formula. For the remainder of this section, we will therefore stick to Euler's solution.

Let us look at some other numbers, not positive integers, for instance:

$$\Gamma(0) = \infty,$$

$$\Gamma(-1) = -\infty,$$

So, $\Gamma(0)$ and $\Gamma(-1)$ yield $\pm\infty$. What about $\frac{1}{2}$? Using *gammal*, we see the following results:

gammal 1 0.5 = 1.8856...
gammal 10 0.5 = 1.7927...
gammal 100 0.5 = 1.7746...
gammal 1000 0.5 = 1.7726...

It will finally approach:

$$\Gamma(0.5) = 1.77267520\dots$$

Is it possible that $\Gamma(0.5)$ yields such a boring number? Well, is it such boring? Look what happens (with one million iterations):

gammal 1000000 0.5 $*$ *gammal* 1000000 0.5 = 3.14159...

That is π! So $\Gamma(0.5) = \sqrt{\pi}$. Not bad! The occurrence of both, e and γ, in Weierstrass' formula already looked somewhat suspicious. It was only a matter of time, when we would meet π in applying the function to some values. In fact, there are many values for which Γ produces a product of $\sqrt{\pi}$ with some fraction. For instance:

$$\begin{aligned}\Gamma\left(\frac{3}{2}\right) &= \frac{1}{2}\sqrt{\pi},\\ \Gamma\left(\frac{5}{2}\right) &= \frac{3}{4}\sqrt{\pi},\\ \Gamma\left(\frac{7}{2}\right) &= \frac{15}{8}\sqrt{\pi},\\ \Gamma\left(\frac{9}{2}\right) &= \frac{105}{16}\sqrt{\pi},\\ &\dots\end{aligned}$$

These results suggest a pattern for odd numbers n. Apparently, $\Gamma(\frac{n}{2})$ yields a product of the form $\frac{k}{2^{(n-1)/2}}\sqrt{\pi}$, where $k = (n-2)(k_{n-2})$.

Can we say more about the factor k? For the odd numbers $1, 3, \dots 11$, k is 1, 3, 15, 105, 945, 10395. These are the *double factorials*, $n!!$, for the odd numbers, *i.e.* the products of all odd numbers $1, 3, .., n$. We, hence, have for odd numbers n:

$$\Gamma\left(\frac{n}{2}\right) = \frac{(n-2)!!}{2^{\frac{n-1}{2}}}\sqrt{\pi} \tag{7.38}$$

which can be implemented in Haskell as

```
gammaho :: Natural → RealN
gammaho 1 = sqrt pi
gammaho n | even n     = error "not an odd number!"
          | otherwise = rff (n − 2) / 2 ∗∗ i
                          ∗ sqrt pi
  where rff = fromIntegral ∘ facfac
        i   = fromIntegral ((n − 1) ‘div‘ 2)
```

The Γ function shows many of such suprising properties. It has been and is still being extensively studied and a lot of relations to other functions, such as the Riemann zeta function, have been found.

But let us now go on to the definition of real binomial coefficients using the Γ function. To this end, we define a new *choose* function, namely:

```
chooser :: Natural → RealN → RealN → RealN
chooser i n k = gammal i (n + 1) /
                  (gammal i (k + 1) ∗ gammal i (n − k + 1))
```

Again, we try this function on integers. For instance:

choose 2 1 = 2
choose 3 1 = 3
choose 3 2 = 3
choose 5 2 = 10
choose 5 3 = 10
choose 7 2 = 21
choose 7 3 = 35

We start with $\binom{2}{1} = 2$, using *chooser*

chooser 1 2 1 = 1.5

Far off. So, again, we increase the number of iterations:

chooser 1 2 1 = 1.5
chooser 10 2 1 = 1.8461...
chooser 100 2 1 = 1.9805...
chooser 1000 2 1 = 1.9980...
chooser 10000 2 1 = 1.9998...

After 10 000 iterations, we come pretty close. Let us try the other examples with 10 000 iterations:

chooser 10000 3 1 = 2.9994...
chooser 10000 3 2 = 2.9994...
chooser 10000 5 2 = 9.9940...
chooser 10000 5 3 = 9.9940...
chooser 10000 7 2 = 10.9790...
chooser 10000 7 3 = 34.9580...

which is fairly close for all these numbers.

The resulting function has been little studied. It is known that many of the binomial identities fail for real numbers. The behaviour for different values of n and k not integers is very complex. We will not go into details here. But we will certainly come back to the Γ-function and its applications later on.

7.9. The Stern-Brocot Tree

Achille Brocot (1817 – 1878) was part of a clockmaker dynasty in Paris started by his father and continuing after his death. The Brocots had a strong emphasis on engineering. Under the pressure of cheap low-quality imports mainly from the USA, they innovated clockmaking with the aim to reduce production cost without equivalent degradation in quality. The constant engineering work manifested in a considerable number of patents hold by family members. The most productive, in terms of engineering, however, was Achille who improved many of his father's inventions and developed new ones. He also introduced a novelty to mathematics, which, surprisingly, has not only practical, but also theoretical value.

In clockmaking, as in machine construction in general, determining the ratio of components to each other, for instance, gear ratios, is a very frequent task. As often in practice, those ratios are not nice and clean, but very odd numbers with many decimal digits. Brocot developed a way to easily approximate such numbers with arbitrary precision and, in consequence, to approximate any real number with arbitrary precision. In the process, he developed yet another way to list all rational numbers.

Brocot's method can be described in terms of finite continued fractions. Recall that we can use lists of the form

$$[n; a, b, c, \dots]$$

to encode continued fractions like

$$n + \cfrac{1}{a + \frac{1}{b + \frac{1}{c + \dots}}}.$$

In contrast to continued fractions we have seen so far, we now look at finite continued fractions that actually result in rational numbers. The process to compute such a continued fraction can be captured in Haskell as:

```
contfracr :: [Ratio] → Ratio
contfracr []       = 0
contfracr [i]      = i
contfracr (i : is) = i + (invert $ contfracr is)
```

Here, *invert* is a function to create the multiplicative inverse of a fraction, *i.e.* $invert(\frac{n}{d}) = \frac{d}{n}$ or in Haskell:

```
invert :: Ratio → Ratio
invert (Q n d) = Q d n
```

As you will see immediately, the expression (*invert* \$ *contfracr is*), corresponds to

$$\frac{1}{contfracr\ is}.$$

The definition above, hence, creates a continued fraction that terminates with the last element in the list.

Now we introduce a simple rule to create from any continued fraction given in list notation two new continued fractions:

```
brocotkids :: [Ratio] → ([Ratio], [Ratio])
brocotkids r = let h  = init r
                   l  = last r
                   s  = length r
                   k1 = h ++ [l + 1]
                   k2 = h ++ [l − 1, 2]
               in if even s then (k1, k2) else (k2, k1)
```

This function yields two lists, k_1 and k_2. They are computed as the initial part of the input list, to which, in the case of k_1, one number is appended, namely the last element of the input list plus 1, or, in the case of k_2, two numbers are appended, namely the last element minus 1 and 2. For the input list $[0, 1]$, which is just 1, for instance, k_1 is $[0, 2]$, which is $\frac{1}{2}$, and k_2 is $[0, 0, 2]$, which is $\frac{1}{\frac{1}{2}} = 2$.

When we compare the paritiy of the length of the lists, we see that k_1 has the same parity as the input list and k_2 has the opposite parity. In particular, if the input list is even, then k_1 is even and k_2 is odd; if it is odd, then k_1 is odd and k_2 is even.

Now, we see for an even list like $[a, b]$ that there is an integer, a, to which the inverse of the second number, b is added. If b grows, then the overall result shrinks. The structure of an odd list is like $[a, b, c]$. Again, the integer a is added to the inverse of what follows in the list. But this time, if c grows, the inverse of c, $\frac{1}{c}$, shrinks and, as such, the value of $\frac{1}{b+1/c}$ grows. Therefore, if the number of elements is even, the value of k_1 is less than the value of the input list and, if it is odd, then the value is greater. You can easily convince yourself that for k_2 this is exactly the other way round. In consequence, the numerical value of the left list returned by *brocotkids* is always smaller than that of the input list and that of the right one is greater.

Using this function, we can now approximate any real number. The idea is that, if the current continued fraction results in a number greater than the number in question, we continue with the left kid; if it is smaller, we continue with the right kid. Here is an implementation:

```
approx :: Natural → RealN → [Ratio]
approx i d = go i [0,1]
  where go :: Natural → [Ratio] → [Ratio]
        go 0 _ = []
        go j r = let k@(Q a b) = contfracr r
                     d' = (fromIntegral a) / (fromIntegral b)
                     (k1,k2) = brocotkids r
                 in if d' ≡ d then [Q a b]
                              else if d' < d then k : go (j − 1) k2
                                             else k : go (j − 1) k1
```

This function takes two arguments. The first, a natural number, defines the number of iterations we want to do. The second is the real number we want to approximate. We start the internal *go* with i and the list $[0,1]$. In *go*, as long as $j > 0$, we compute the rational number that corresponds to the input list; then we compute the corresponding real number. If the number we computed this way equals the input (*i.e.* the input is rational), we are done. Otherwise, if it is less than the input, we continue with k_2; if it is greater, we continue with k_1.

The function yields the whole trajectory whose last number is the best approximation with n iterations. The result of *approx* 10 *pi*, for instance is:

$$1,2,3,4,\frac{7}{2},\frac{10}{3},\frac{13}{4},\frac{16}{5},\frac{19}{6},\frac{22}{7}.$$

The last fraction $\frac{22}{7}$ is approximately 3.142857, which still is a bit away from 3.141592. We reach 3.1415 with *approx* 25 *pi*, for which the last fraction is $\frac{333}{106} = 3.141509$. This way, we can come as close to π as we wish.

Since, with the *brocotkids* function, we always create two follow-ups for the input list, we can easily define a binary tree where, for each node, k_1 is the left subtree and k_2 is the right subtree. This tree, in fact, is well-known and is called *Stern-Brocot tree* in honour of Achille Brocot and Moritz Stern, the German number theorist we already know from the discussion of the Calkin-Wilf tree.

The Stern-Brocot tree can be defined as

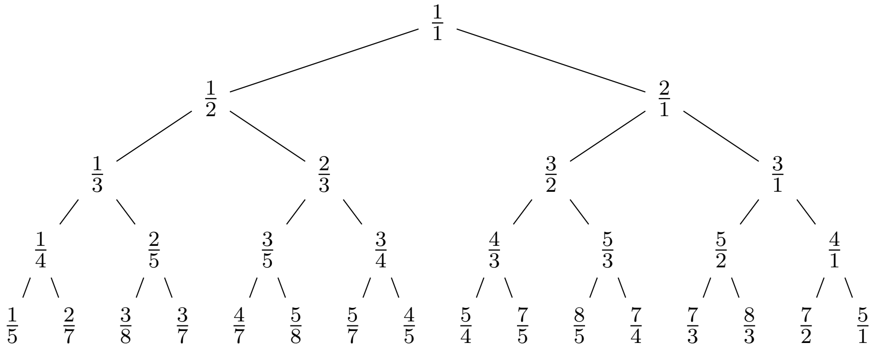
```
type SterBroc = Tree [Ratio]
sterbroc :: Zahl → [Ratio] → SterBroc
sterbroc i r | i ≡ 0     = Node r []
             | otherwise = let (k1, k2) = brocotkids r
                           in Node r [sterbroc (i − 1) k1,
                                      sterbroc (i − 1) k2]
```

The function *sterbroc* takes an integer argument to define the number of generations we want to create and an initial list of *Ratio*. If we have exhausted the number of generations, we create the final *Node* without kids. Otherwise, we create the *brocotkids* and continue with *sterbroc* on k_1 and k_2. If we start with a negative number, we will generate infinitely many generations.

We can now convert the continued fractions in the nodes to fractions by *fmap*ing *contfracr* on them. Here is a function that creates the Stern-Brocot Tree from root node $[0, 1]$ labled with fractions:

```
sterbrocTree :: Zahl → Tree Ratio
sterbrocTree i = fmap contfracr (sterbroc i [0, 1])
```

For the first five generations, this tree is



As you can see at once, this tree has many properties in common with the Calkin-Wilf tree. First and trivially, the left kid of a node k is less than k and the right kid of the same node is greater than k. The left-most branch of the tree contains all fractions with 1 in the numerator like $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}$ and so on. The right-most branch contains the integers $\frac{1}{1}, \frac{2}{1}, \frac{3}{1}, \frac{4}{1}$ and so on.

Furthermore, the product of each generation is 1. For instance, $\frac{1}{1} = 1$, $\frac{1}{2} \times \frac{2}{1} = 1$, $\frac{1}{3} \times \frac{2}{3} \times \frac{3}{2} \times \frac{3}{1} = 1$ and so on. In fact, we see in each generation the same fractions we would also see in the Calkin-Wilf tree. The order of the fraction, however, is different. More precisely, the order of the inner fractions differs, since, as we have seen, the left-most and right-most numbers are the same.

We could hence ask the obvious question: how can we permute the generations of the

Stern-Brocot tree to obtain the generations of the Calkin-Wilf tree and vice versa? Let us look at an example. The 4^{th} generation of the Calkin-Wilf tree is *getKids* 4 (*calWiTree* 4 (Q 1 1)):

$$\frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, \frac{4}{1}.$$

The 4^{th} generation of the Stern-Brocot tree is *getKids* 4 (*sterbroctree* 4):

$$\frac{1}{4}, \frac{2}{5}, \frac{3}{5}, \frac{3}{4}, \frac{4}{3}, \frac{5}{3}, \frac{5}{2}, \frac{4}{1}.$$

We see that only some fractions changed their places and the changes are all direct swaps, such that the second position in the Calkin-Wilf tree changed with the fifth position and the fourth position changed with the seventh position. The other positions, the first, third, sixth and eighth, remain in their place. We could describe this in cyclic notation, using indexes from 0 – 7 for the eight positions:

$$(1, 4)(3, 6).$$

In other words, we represent the generations as arrays with indexes 0 – 7:

	0	1	2	3	4	5	6	7
Calkin-Wilf	$\frac{1}{4}$	$\frac{4}{3}$	$\frac{3}{5}$	$\frac{5}{2}$	$\frac{2}{5}$	$\frac{5}{3}$	$\frac{3}{4}$	$\frac{4}{1}$
Stern-Brocot	$\frac{1}{4}$	$\frac{2}{5}$	$\frac{3}{5}$	$\frac{3}{4}$	$\frac{4}{3}$	$\frac{5}{3}$	$\frac{5}{2}$	$\frac{4}{1}$

So, what is so special about the indexes 1, 3, 4 and 6 that distinguishes them from the indexes 0, 2, 5 and 7? When we represent these numbers in binary format with leading zeros, so that all binary numbers have the same length, we have

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111

When we look at the indexes whose fractions do not change, *i.e.* 0, 2, 5 and 7, we see one property that they all have in common: they are all symmetric. That is, when we reverse the bit strings, we still have the same number. 0 = 000 reversed is still 000 = 0; 2 = 010 reversed is still 010 = 2; 5 = 101 reversed is still 101 = 5 and 7 = 111 reversed is still 111 = 7. 1 = 001 reversed, however, is 100 = 4 and vice versa and 3 = 011 reversed is $110 = 6$. This corresponds exactly to the permutation $(1,4)(3,6)$ and is an instance of a bit-reversal permutation.

Let us try to implement the bit-reversal permutation. First we implement the bit-reverse of the indexes. To do so, we first need to convert the decimal index into a binary number; then we add zeros in front of all binary numbers that are shorter than the greatest number; then we simply reverse the lists of binary digits, remove the leading zeros and convert back to decimal numbers. This can be nicely expressed by the function

```
bitrev :: Int → Int → Int
bitrev x = fromIntegral ∘ fromBinary ∘
           cleanz ∘ reverse ∘ fillup x 0 ∘
           toBinary ∘ fromIntegral
```

where *fillup* is defined as

```
fillup :: Int → Int → [Int] → [Int]
fillup i z is | length is ≡ i = is
              | otherwise     = fillup i z (z : is)
```

and *cleanz* as

```
cleanz :: [Int] → [Int]
cleanz []       = []
cleanz [0]      = [0]
cleanz (0 : is) = cleanz is
cleanz is       = is
```

To apply this, we first have to calculate the size of the greatest number in our set in binary format. If we assume that we have a list of consecutive numbers from $0 \dots n-1$, then the size of the greatest number is just $\log_2 n$, the binary logarithm of n. For $n = 8$, for instance, this is 3. With this out of the way, we can define a bit reversal of the indexes of any set $[a]$ as:

```
idxbitrev :: [a] → [Int]
idxbitrev xs = let l = fromIntegral $ length xs
                   x = round $ logBase 2 (fromIntegral l)
               in [bitrev x i | i ← [0 . . l − 1]]
```

and use this function to permute the original input list:

```
bitreverse :: [a] → [a]
bitreverse xs = go xs (idxbitrev xs)
  where go _ []         = []
        go zs (p : ps) = zs !! p : go zs ps
```

Applied on the list $[0,1,2,3,4,5,6,7]$, we see exactly the $(1,4)(3,6)$ permutation we saw above, namely $[0,4,2,6,1,5,3,7]$. Applied on a generation from the Calkin-Wilf tree, we see the corresponding generation from the Stern-Brocot tree.

Let $t = \mathit{calWiTree}\ (-1)\ (1 \,\%\, 1)$, we see for

getKids 3 *t*	$\frac{1}{3}, \frac{3}{2}, \frac{2}{3}, \frac{3}{1}$
bitreverse (*getKids* 3 *t*)	$\frac{1}{3}, \frac{2}{3}, \frac{3}{2}, \frac{3}{1}$
getKids 4 *t*	$\frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, \frac{4}{1}$
bitreverse (*getKids* 4 *t*)	$\frac{1}{4}, \frac{2}{5}, \frac{3}{5}, \frac{3}{4}, \frac{4}{3}, \frac{5}{3}, \frac{5}{2}, \frac{4}{1}$

Since the generations of the Stern-Brocot tree are nothing but permutations of the Calkin-Wilf tree, we can derive a sequence from the Stern-Brocot tree that lists all rational numbers. We create this sequence in exaclty the same way we did for the CalkinWilf tree, namely

```
enumQsb :: [Ratio]
enumQsb = go 1 $ sterbrocTree (−1)
  where go i t = getKids i t ++ go (i + 1) t
```

The numerators of the sequence derived in this way from the Calkin-Wilf tree equal the well-known Stern sequence. Is there another well-known sequence that is equivalent to the numerators of the Stern-Brocot tree sequence? Let us ask the On-line Encyclopedia with the first segment of that sequence generated by *map numerator* (*take* 20 *enumQsb*):

$$1, 1, 2, 1, 2, 3, 3, 1, 2, 3, 3, 4, 5, 5, 4, 1, 2, 3, 3, 4, 5, 5, 4, 5, 7.$$

The Encyclopedia tells us that this is the numerators of the *Farey sequence.* This sequence, named for British geologist John Farey (1766 – 1826), has a lot of remarkable properties. The Farey sequence of n lists all fractions in canoncial form between 0 and 1, usually included, with a denominator less than or equal to n. For instance, the Farey sequence of 1, designated F_1 just contains $0, 1$; F_2 contains $0, \frac{1}{2}, 1$; F_3 contains $0, \frac{1}{3}, \frac{2}{3}, 1$ and so on.

A direct way to implement this could be to combine all numbers from $0 \dots n$ in the numerator with all numbers $1 \dots n$ in the denominator that are smaller than 1 and to sort and *nub* the resulting list, like this:

```
farey2 :: Natural → [Ratio]
farey2 n = sort (nub $ filter (⩽ 1) $
                     concatMap (λx → map (x%) [1 .. n]) [0 .. n])
```

With this approach, we create a lot of fractions that we do not need and that we filter out again afterwards. A more interesting approach, also in the light of the topic of this section, is the following:

```
farey :: Natural → [Ratio]
farey n = 0 : sort (go 1 $ sterbrocTree (−1))
  where go k t = let g = getKids k t
                     l = filter fltr g
                 in if null l then l else l ++ go (k + 1) t
        fltr k  = k ⩽ 1 ∧ n ⩾ denominator k
```

Here, we iterate over the generations of the Stern-Brocot tree removing the fractions that are greater than 1 or have a denominator greater n. When we do not get results anymore, *i.e.* all denominators are greater than n, we are done.

Let us try this algorithm on some numbers:

$$F_4 = \left\{0, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, 1\right\} \tag{7.39}$$

$$F_5 = \left\{0, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, 1\right\} \tag{7.40}$$

$$F_6 = \left\{0, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, 1\right\} \tag{7.41}$$

We see some interesting properties. First and this should be obvious, we see n as a denominator in sequence F_n exaclty $\varphi(n)$ times. For F_6, for instance, we could create the fractions $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$ and $\frac{5}{6}$. The fractions $\frac{2}{6} \dots \frac{4}{6}$, however, are not in canonical form, since the numerators $2 \dots 4$ all share divisors with 6. Since there are $\varphi(n)$ numerators that do not share divisors with n, there are only $\varphi(n)$ fractions less than 1 whith n in the denominator.

Another property is that, for two consecutive fractions in the Farey sequence, $\frac{a}{b}$ and $\frac{c}{d}$, the cross products ad and cb are consecutive integers. In again F_6, for the fractions $\frac{1}{6}$ and $\frac{1}{5}$, the cross products, trivially, are 5 and 6. More interesting are the fractions $\frac{3}{5}$ and $\frac{2}{3}$ whose cross products are $3 \times 3 = 9$ and $5 \times 2 = 10$.

Even further, for any three consecutive fractions in a Farey sequence, the middle one, called the mediant fraction, can be calculated from the outer ones as $\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$. For instance in F_6:

$$\frac{1+1}{6+4} = \frac{2}{10} = \frac{1}{5}, \tag{7.42}$$

$$\frac{2+3}{5+5} = \frac{5}{10} = \frac{1}{2} \tag{7.43}$$

and

$$\frac{3+5}{4+6} = \frac{8}{10} = \frac{4}{5}. \tag{7.44}$$

This property can be used to compute F_{n+1} from F_n. We just have to insert those mediant fractions of two consecutive fractions in F_n, for which the denominator is $n+1$. In F_6 we would insert

$$\frac{0+1}{1+6}, \frac{1+1}{4+3}, \frac{2+1}{5+2}, \frac{1+3}{2+5}, \frac{2+3}{3+4}, \frac{5+1}{6+1}$$

resulting in

$$F_7 = \left\{0, \frac{1}{7}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{2}{7}, \frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{1}{2}, \frac{4}{7}, \frac{3}{5}, \frac{2}{3}, \frac{5}{7}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{6}{7}, 1\right\} \tag{7.45}$$

We can implement this as

```
nxtFarey :: Natural → [Ratio] → [Ratio]
nxtFarey n []  = []
nxtFarey n [r] = [r]
nxtFarey n (a : b : rs) | denominator a +
                          denominator b ≡ n = nxtFarey n (a : x : b : rs)
                        | otherwise         = a : nxtFarey n (b : rs)
  where x = let n1 = numerator a
                n2 = numerator b
                d1 = denominator a
                d2 = denominator b
            in (n1 + n2) % (d1 + d2)
```

In fact, we can construct the Stern-Brocot tree by means of mediant fractions. The outer fractions, in this algorithm are the predecessors of the current node, namely the direct predecessor and either the predecessor of the predecessor or the sibling of that node. For instance, the second node in the third generation is $\frac{2}{3}$. Its kids are $\frac{3}{5}$ and $\frac{3}{4}$. $\frac{3}{5}$ is $\frac{2+1}{3+2}$ and, thus, the sum of $\frac{2}{3}$ and its predecessor; $\frac{4}{3}$, however, is $\frac{2+1}{3+1}$ and, hence, the sum of the $\frac{2}{3}$ and the predecessor of its predecessor.

The question now is how to bootstrap this algorithm, since the root node does not have predecessors. For this case, we imagine two predecessors, namely the fractions $\frac{0}{1}$ and $\frac{1}{0}$, the latter of which, of course, is not a proper fraction. The assumption of such nodes, however, helps us derive the outer branches, where, on the left side, the numerator does not change, hence is constructed by addition with 0, and, on the right side, the denominator does not change and is likewise constructed by addition with 0.

We implement this as

```
mSterbroctree :: Zahl → Natural → Natural →
                        Natural → Natural → Ratio → Tree Ratio
mSterbroctree 0 _ _ _ _ r = Node r [ ]
mSterbroctree n a b c d r =
  let rn = numerator r
      rd = denominator r
      k1 = (a + rn) % (b + rd)
      k2 = (c + rn) % (d + rd)
  in  if k1 < k2
      then Node r [mSterbroctree (n − 1) a b rn rd k1,
                   mSterbroctree (n − 1) c d rn rd k2]
      else  Node r [mSterbroctree (n − 1) c d rn rd k2,
                   mSterbroctree (n − 1) a b rn rd k1]
```

Note that we have to use two pairs of natural numbers instead of two fractions to encode the predecessors. This is because we have to represent the imagined predecessor $\frac{1}{0}$, which is not a proper fraction. Finally, we check for the smaller of the resulting numbers k_1 and k_2 to make sure that the smaller one always goes to the left and the greater to the right. This implementation now gives exactly the same tree as the implementation using continued fractions introduced at the beginning of the section.

7.10. Field Extension

A very common activity of mathematicians is solving equations. They usually solve equations with coefficients of a certain type of numbers (like integers or rationals) assuming that the solution is of that same number type. A typical example is Diophantine equations, named after the late-antique mathematician Diophantus of Alexandria who lived in the third century. He studied equations and is the first mathematician to be known to have introduced abstract symbols for numbers. Diophantine equations operate over the integers. The known values, *i.e.* the coefficients, as well as the unknown values, *i.e.* the solutions, must be integers. The most famous result from the study of Diophantine equations is perhaps the proof of Fermat's Last Theorem, which states that there are no solutions for $z > 2$ in equations of the form

$$a^z + b^z = c^z \tag{7.46}$$

where a, b, c and z are all integers. Fermat scribbled his conjecture in the margin of his copy of Diophantus' "Arithmetica". It turned into his last *theorem* only when Andrew Wiles proved it in the 90ies of the 20^{th} century using concepts that went far beyond the knowledge of Fermat and his contemporaries.

In modern times, equations are typically studied in a *field* and you might remember that a field is a structure defined over a set of numbers with two operations. Both operations establish an *Abelian group* with that set of numbers where one operation (called *multiplication*) distributes over the other (called *addition*). More formally, a field is defined as a structure

$$(S, +, \times),$$

where S is the set of numbers, "+" the addition operation and "×" multiplication. For both operations, the following properties must hold:

1. **Closure**: for all $a, b \in S : a \circ b \in S$.
2. **Associativity**: $a \circ (b \circ c) = (a \circ b) \circ c = a \circ b \circ c$.
3. **Identity**: there is exactly one element $e \in S$, called the identity, such that for all $a \in S : a \circ e = e \circ a = a$.
4. **Invertibility** for each element $a \in S$, there is an element a', such that $a \circ a' = e$.
5. **Commutativity** $a \circ b = b \circ a$.

Properties 1 – 4, as you will have realised, are just the group laws. Property 5, commutativity, makes the group *Abelian.*

Furthermore, multiplication **distributes** over addition, i.e.

$$a \times (b + c) = ab + ac.$$

When all these properties hold, then we have a field. We have already seen that $\mathbb{Q}$, the rational numbers, is a field. Historically, this field was important for the theory of solving equations. For the special case of linear equations, that is equations without exponents, rational coefficients lead to rational solutions. The reason is that the operations we need to solve linear equations are just the four arithmetic operations addition, subtraction, multiplication and division. A simple equation of the form

$$ax + b = 0 \tag{7.47}$$

is solved by first subtracting b from both sides of the equation and then dividing both sides by a leading to

$$x = -\frac{b}{a}. \tag{7.48}$$

Note that the solutions are not necessarily integers, like in Diophantine equations, since not every integer has a multiplicative inverse. In other words, integers do not constitute a group over multiplication and integers, therefore, do not form a field. In the field $\mathbb{Q}$ of rational numbers, however, there is a solution (and exactly one solution) that lies within that field.

But, of course, there are equations that cannot be solved by applying the four fundamental arithmetic operations alone, quadratic equations, for instance:

$$x^2 - 2 = 0. \tag{7.49}$$

We proceed like for the linear equation: we subtract -2 on both sides and then, instead of dividing by something, we take the square root leading to

$$x = \sqrt{2}. \tag{7.50}$$

Unfortunately, $\sqrt{2}$ is, as we already know, not in the field $\mathbb{Q}$. It is irrational. We can of course redefine the field in which we started to solve this equation in the first place assuming $\mathbb{R}$ instead of $\mathbb{Q}$. But that is a sloppy solution. A typical question for a mathematician is: what is the *smallest* field comprising both, the coefficients and the solutions of this kind of equations? A possible answer is: the field $\mathbb{Q}$ with the solution $\sqrt{2}$ added to it. That is, we *extend* the field $\mathbb{Q}$ by *adjoining* $\sqrt{2}$. We should then get a new field, called $\mathbb{Q}(\sqrt{2})$, of which the original field $\mathbb{Q}$ is a subfield, *i.e.*

$$\mathbb{Q} \subset \mathbb{Q}\left(\sqrt{2}\right).$$

What does this new field look like? Of course, we cannot just extend the underlying set, such like:

$$S = \left\{0, 1, \frac{1}{2}, 2, \frac{1}{3}, 3, \dots, \infty, \sqrt{2}\right\}.$$

This, obviously, would not lead to a field, since not for every $a \in S$ $a\sqrt{2} \in S$. $2 \times \sqrt{2}$, for instance, is not in the field; $\frac{1}{2} \times \sqrt{2}$, too, is not in the field and so on. In fact, for almost no $a \in S$, closure is fulfilled. It is fulfilled only for 0, the identity and $\sqrt{2}$ itself, since $0 \times \sqrt{2} = 0 \in S$, $1 \times \sqrt{2} = \sqrt{2} \in S$ and $\sqrt{2} \times \sqrt{2} = 2 \in$S.

So, we need a better approach. The following formalism defines the *smallest* field that contains $\mathbb{Q}$ and the square root of any number $r \in \mathbb{Q}$. We first define a new number type consisting of a tuple (a, b), for $a, b \in \mathbb{Q}$. The natural interpretation of this tuple is

$$(a, b) = a + b\sqrt{r}. \tag{7.51}$$

The set underlying the field $\mathbb{Q}(\sqrt{r})$, thus, consists of the numbers

$$\left\{a + b\sqrt{r} | a, b \in \mathbb{Q}\right\},$$

with r being a constant rational number, *i.e.* $r \in \mathbb{Q}$. If you want to look at the concrete numbers, you may reformulate this in terms of Haskell list comprehension:

```
[(fromRational a) +
 (fromRational b) ∗ (sqrt r) | a ← enumQ,
                               b ← enumQ]
```

Do not be confused by the fact that, in Haskell, we have to convert a and b to real numbers. Haskell has no built-in notion of field extension. Numbers are either rational or real. Since *sqrt r* is *RealN*, we have to convert everything to *RealN*. The result, however, shows what the numbers in the new field look like "in reality", whatever that is supposed to mean.

Now we define the arithmetic operations in a way that fulfils the group properties. First, addition is

$$(a, b) + (c, d) = (a + c, b + d). \tag{7.52}$$

That is easy and, in fact, follows from basic properties of addition in the field $\mathbb{Q}$. If we have something like $a + bx + c + dx$, we usually simplify to $a + c + (b + d)x$. That is just the same as we did above.

Multiplication is a bit more complicated. Let us first ask, how the product of two expressions of the form $a + bx$ and $c + dx$ looks like:

$$(a + bx)(c + dx) = ac + adx + bcx + bdx^2.$$

Since, x in our case is the square root of r, $x^2 = \sqrt{r} \times \sqrt{r} = r$ and we, hence, get

$$ac + rbd + (ad + bc)\sqrt{r}.$$

From this we can derive the general rule

$$(a, b)(c, d) = (ac + rbd, ad + bc), \tag{7.53}$$

where r is the number, whose square root is adjoint to our base field. For $\mathbb{Q}(\sqrt{2})$, this is

$$(a, b)(c, d) = (ac + 2bd, ad + bc). \tag{7.54}$$

This construction of the field extension guarantees that for any addition and any multiplication of two elements in this new field, the result, again, is an element of this field. The rules also guarantee associativity, as you may easily convince yourself. But what about the identities of addition and multiplication?

In general, a rationl number a is, in the new field, represented as $(a, 0)$. This is easy to see, because $(a, 0) = a + 0 \times \sqrt{r} = a$. Since the additive identity is 0 in $\mathbb{Q}$, the identity should be $(0, 0)$. We just follow rule 7.52 to prove that:

$$(a + b) + (0, 0) = (a + 0, b + 0) = (a + b) \quad \square. \tag{7.55}$$

What about the multiplicative identity? We expect it to be the representation of 1 in the new field, which is $(1, 0)$, since $1 + 0\sqrt{r} = 1 + 0 = 1$. Let us check:

$$(a + b)(1, 0) = (1a + 2b \times 0, a \times 0 + 1b) = (a, b), \tag{7.56}$$

which, indeed, fulfils the identity property. $\square$

The next question is how the inverse will look like in the new field. For the additive inverse that is not difficult to answer. Since, for any number (a, b), the additive inverse $-(a, b)$ should fulfil the property $(a, b) + -(a, b) = (0, 0)$, the inverse must therefore be

$$-(a, b) = (-a, -b). \tag{7.57}$$

We can check this quickly using again 7.52:

$$(a, b) + (-a, -b) = (a - a, b - b) = (0, 0). \quad \square \tag{7.58}$$

Concerning multiplication, which, as usual, is a bit more complicated than addition, the inverse is

$$(a, b)^{-1} = \left(\frac{a}{a^2 - rb^2}, -\frac{b}{a^2 - rb^2}\right). \tag{7.59}$$

Here is the proof using 7.53:

$$(a, b)\left(\frac{a}{a^2 - rb^2}, -\frac{b}{a^2 - rb^2}\right) = \left(\frac{a^2}{a^2 - rb^2} - \frac{rb^2}{a^2 - rb^2}, \frac{ab}{a^2 - rb^2} - \frac{ab}{a^2 - rb^2}\right). \tag{7.60}$$

The scary looking formula on the right-hand side of the equation can be simplified. The first component is

$$\frac{a^2}{a^2 - rb^2} - \frac{rb^2}{a^2 - rb^2},$$

which can be reduced to one fraction, since the denominators are equal:

$$\frac{a^2 - rb^2}{a^2 - rb^2} = 1.$$

We see a fraction with identical numerator and denominator. The fraction, hence, can be further reduced to 1.

The second component is

$$\frac{ab}{a^2 - rb^2} - \frac{ab}{a^2 - rb^2} = 0.$$

We finally get

$$(a, b)\left(\frac{a}{a^2 - rb^2}, -\frac{b}{a^2 - rb^2}\right) = (1, 0), \tag{7.61}$$

which proves that the beast in 7.59 fulfils the invertibility property. □

The final piece in showing that $\mathbb{Q}(\sqrt{r})$ is indeed a field considering the rules 7.52 and 7.53 is distributivity. Distributivity requires that

$$(a, b)((c, d) + (e, f)) = (a, b)(c, d) + (a, b)(e, f). \tag{7.62}$$

Multiplying the left side out, we get

$$(ac + rbd, ad + bc) + (ae + rbf, af + be),$$

which, when added, is

$$(ac + rbd + ae + rbf, ad + bc + af + be).$$

We show that this is true by multiplying

$$(a + b\sqrt{r})(c + d\sqrt{r} + e + f\sqrt{r}).$$

We regroup the second part:

$$c + e + d\sqrt{r} + f\sqrt{r}$$

and distribute, first a:

$$ac + ae + ad\sqrt{r} + af\sqrt{r}$$

and then $b\sqrt{r}$:

$$cb\sqrt{r} + eb\sqrt{r} + d\sqrt{r}b\sqrt{r} + f\sqrt{r}b\sqrt{r}.$$

This second term simplifies to

$$cb\sqrt{r} + eb\sqrt{r} + bdr + fbr.$$

Now we bring the two terms together and get

$$ac + ae + rbd + rbf + (ad + bc + af + be)\sqrt{r} = (ac + ae + rbd + rbf, ad + bc + af + be)$$

as desired. □

We have defined the smallest field that extends $\mathbb{Q}$ by adjoining $\sqrt{r}$ for any rational number r. This is nice, because it allows us to add the square roots of any rational number using the same recipe. We can even go further and extend the extended field by adjoining the square roots of square roots on top of the extension already containing the square roots, *i.e.*:

$$\mathbb{Q}(\sqrt{r}, \sqrt[4]{r}).$$

We can go still further and add the square roots of the square roots of the square roots:

$$\mathbb{Q}(\sqrt{r}, \sqrt[4]{r}, \sqrt[8]{r})$$

and then the square roots of the square roots of the square roots of the square roots and so on *ad infinitum.* There are indeed classic problems, as we will see later, that can be solved in exactly this field: $\mathbb{Q}$ extended by the n^{th}-roots, where n is any power of 2.

Extensions resulting from building extensions on top of extensions are sometimes called *towers of fields* where one field is put on the top of another field yielding a batch of pancakes that slowly grows higher and higher. This technique is often used in algebra, more specifically in *Galois Theory*, to study equations of higher degrees. For instance, the equation

$$x^4 - 4x^3 - 4x^2 + 8x - 2 = 0 \tag{7.63}$$

has four solutions, namely

$$\begin{aligned}
x_1 &= 1 + \sqrt{2} + \sqrt{3 + \sqrt{2}} \\
x_2 &= 1 + \sqrt{2} - \sqrt{3 + \sqrt{2}} \\
x_3 &= 1 - \sqrt{2} + \sqrt{3 - \sqrt{2}} \\
x_4 &= 1 - \sqrt{2} - \sqrt{3 - \sqrt{2}}
\end{aligned}$$

We can build a tower of extensions of $\mathbb{Q}$ by stepwise adjoining the irrational components of the solutions:

$$\begin{aligned}
&\mathbb{Q} \\
&\mathbb{Q}\left(\sqrt{2}\right) \\
&\mathbb{Q}\left(\sqrt{2}, \sqrt{3 + \sqrt{2}}\right) \\
&\mathbb{Q}\left(\sqrt{2}, \sqrt{3 + \sqrt{2}}, \sqrt{3 - \sqrt{2}}\right)
\end{aligned}$$

The interesting question presents itself whether it is possible to build a tower from $\mathbb{Q}$ to $\mathbb{R}$. It is indeed possible. But it is not trivial. $\mathbb{R}$ is in fact much bigger than $\mathbb{Q}$. How much bigger, we will soon see.

The difficulty arising in building that tower is that we need different definitions for different things we adjoin to $\mathbb{Q}$ and its extensions. Until now, we have only looked at square roots. But how to add n-roots where n is not a power of two? $\mathbb{Q}(\sqrt[3]{r})$, for instance, can not be represented by the formulas above. This is because $\sqrt[3]{r} \times \sqrt[3]{r}$ is not a rational number and is therefore not in the field. The *degree* of this extension is not the same as that of $\mathbb{Q}(\sqrt{r})$. The degree of $\mathbb{Q}(\sqrt{r})$ is 2, since it can be represented by a pair of numbers (a, b). $\mathbb{Q}(\sqrt[3]{r})$, however, cannot be represented by a pair; a triple is needed, namely the triple

$$(a, b, c) = a + b\sqrt[3]{r} + c\sqrt[3]{r^2}.$$

The degree of $\mathbb{Q}(\sqrt[3]{r})$ is therefore 3.

What does the field $\mathbb{Q}(\sqrt[3]{r})$ look like? For addition, it looks very similar to the field $\mathbb{Q}(\sqrt{r})$. The addition rule is

$$(a, b, c) + (d, e, f) = (a + d, b + e, c + f). \tag{7.64}$$

The additive identity, trivially, is $(0, 0, 0)$. The inverse $-(a, b, c)$ is $(-a, -b, -c)$.

Harder, however, is multiplication. Let us investigate the multiplication rule. We try to multiply two numbers in the new field

$$(a, b, c)(d, e, f).$$

This corresponds to the expression

$$(a + b\sqrt[3]{r} + c\sqrt[3]{r^2})(d + e\sqrt[3]{r} + f\sqrt[3]{r^2}).$$

We distribute the first sum term by term over the second sum. Distributing a gives

$$ad + ae\sqrt[3]{r} + af\sqrt[3]{r^2};$$

Distributing $b\sqrt[3]{r}$ gives

$$bd\sqrt[3]{r} + be\sqrt[3]{r}\sqrt[3]{r} + bf\sqrt[3]{r}\sqrt[3]{r^2}$$

and distributing $c\sqrt[3]{r^2}$ gives

$$cd\sqrt[3]{r^2} + ce\sqrt[3]{r^2}\sqrt[3]{r} + cf\sqrt[3]{r^2}\sqrt[3]{r^2}.$$

Now, we represent the roots as fractional exponents and get:

$$ad + aer^{\frac{1}{3}} + afr^{\frac{2}{3}}$$

for the first component,

$$bdr^{\frac{1}{3}} + ber^{\frac{1}{3}}r^{\frac{1}{3}} + bfr^{\frac{1}{3}}r^{\frac{2}{3}}$$

for the second and

$$cdr^{\frac{2}{3}} + cer^{\frac{2}{3}}r^{\frac{1}{3}} + cfr^{\frac{2}{3}}r^{\frac{2}{3}}$$

for the third. We can simplify the second component to

$$bdr^{\frac{1}{3}} + ber^{\frac{2}{3}} + bfr$$

and the third to

$$cdr^{\frac{2}{3}} + cer + cfr^{\frac{4}{3}}.$$

The last term in this expression is not very nice. It, apparently, introduces a new element that we do not yet know. However, we can transform it:

$$r^{\frac{4}{3}} = r^{\frac{3}{3}+\frac{1}{3}} = rr^{\frac{1}{3}}$$

resulting in a product of two elements we do know already, namely r, which is a rational number, and $r^{\frac{1}{3}}$, which is just $\sqrt[3]{r}$. The last component, hence, is

$$cdr^{\frac{2}{3}} + cer + cfrr^{\frac{1}{3}}.$$

We will now group the terms in the components according to their exponents. First the terms without exponent

$$ad + bfr + cer,$$

then those with exponent $\frac{1}{3}$:

$$aer^{\frac{1}{3}} + bdr^{\frac{1}{3}} + cfrr^{\frac{1}{3}}$$

and, finally, those with exponent $\frac{2}{3}$:

$$afr^{\frac{2}{3}} + ber^{\frac{2}{3}} + cdr^{\frac{2}{3}}.$$

When we convert the exponents back to roots, we see that we have three groups: one consisting of rational numbers only, one consisting of rational numbers multiplied by $\sqrt[3]{r}$ and, finally, one consisting of rational numbers multiplied by $\sqrt[3]{r^2}$. We conclude that the multiplication formula in this field is

$$(a, b, c)(d, e, f) = (ad + bfr + cer, ae + bd + cfr, af + be + cd), \tag{7.65}$$

where r is the rational number, whose third root was adjoint to $\mathbb{Q}$.

The multiplicative identity should be $(1, 0, 0)$ and, indeed, $(a, b, c)(1, 0, 0)$ gives according to 7.65:

$$(a + 0 + 0, 0 + b + 0, 0 + 0 + c) = (a, b, c). \quad \square$$

What, however, is the multiplicative inverse? Well, answering this questions corresponds to solving the equation

$$ad + bfr + cer + (ae + bd + cfr)\sqrt[3]{r} + (af + be + cd)\sqrt[3]{r^2} = 1. \tag{7.66}$$

Solving such equations is a major topic of the next part. With the techniques we have at our disposal now, this is not easy. We will come back to that question later. Anyway, what should be clear from the exercise is that it is possible to extend the field $\mathbb{Q}$ step by step including always more irrational numbers until we reach $\mathbb{R}$. But this process is not trivial. It involves a lot of algebra. It is a true Tower of Babel. And, until here, we have only looked at irrational numbers that are roots of rational numbers. We have not yet discussed how to extend fields by *transcendental numbers*, *i.e.* numbers that are not roots of rational numbers and, even further, do not appear as solutions of equations with rational coefficients at all. Our friends π and e are examples of such numbers.

7.11. The Continuum

So. How many real numbers are there? As before, we will try to *count* the real numbers by building a set of tuples (r, n) for all numbers $r \in \mathbb{R}$ and $n \in \mathbb{N}$. In order to do this, we need to first establish a sequence of real numbers. We start by investigating the real numbers in the range $0 \dots 1$. We construct a sequence of real numbers in this interval. We start by creating some sequence like the following:

$$\begin{gathered} 0.10001\dots \\ 0.01001\dots \\ 0.00101\dots \\ 0.00011\dots \\ 0.00001\dots \\ \dots \end{gathered}$$

We now have an infinite sequence of infinite sequences of digits. Can we enumerate this sequence and assign natural numbers to each single real number to count the whole sequence? There is an issue. For any given sequence, we can easily construct a new number, not yet in the sequence, but in the same interval (the numbers $0 \ldots 1$). The method is called Cantor's (second) *diagonal argument*. It goes like this: For each number in the sequence, from the first number take the first digit, from the second number take the second digit, from the third number take the third digit and so on. Since the numbers in the sequence are distinct, the new number, constructed in this way, is different from all other numbers in the sequence. For instance:

$$
\begin{gathered}
0.\mathbf{1}0001\ldots \\
0.0\mathbf{1}001\ldots \\
0.00\mathbf{1}01\ldots \\
0.000\mathbf{1}1\ldots \\
0.0000\mathbf{1}\ldots \\
\ldots \\
0.\mathbf{11111}\ldots
\end{gathered}
$$

Note that the sequence above was not particularly designed to hold for this method. As long as the numbers are irrational, *i.e.* each of them consists of an infinite non-repeating sequence of digits, we will, following the method, always construct a number that is not yet in the list.

The point is that we can do this with any sequence of irrational numbers one may come up with. In consequence, when we construct an enumeration of the real numbers, as we did for rational numbers using the Calkin-Wilf or the Stern-Brocot tree, there is always a number that we can introduce between any two numbers in the sequence. Any possible sequence of the real numbers, hence, is necessarily incomplete. We, therefore, arrive at the strange conclusion that $|\mathbb{R}| \neq |\mathbb{N}|$ or, in other words, $\mathbb{R}$ is not *countable*.

Cantor says that the sets are both infinite, but they are infinite in different ways: $\mathbb{N}$ is countably infinite, while $\mathbb{R}$ is *uncountably* infinite. There are thus different infinite cardinalities. That of $\mathbb{N}$ is $\aleph_0$; that of other sets may be $\aleph_1, \aleph_2, \aleph_3, \ldots$ leading to a whole new universe of numbers that express different ways of being infinite.

Cantor conjectured that the cardinality of $\mathbb{R}$ is the cardinality of the *powerset* of $\mathbb{N}$, which is, as you may remember, 2^n for a set with n elements. The cardinality of $\mathbb{R}$, would then be $2^{\aleph_0}$.

This makes a lot of sense, when you consider the diagonal method above. Indeed, when we created the powerset of a given set, we used binary numbers to encode the presence or absence of a given element in one of the sets in the powerset. For the set $S = \{a, b, c\}$, the number 100_2 would encode the set $\{a\} \in P(S)$. Now, when you consider that S is infinite, you have an infinite sequence of such numbers and, as we did above illustrating the diagonal argument, we can introduce for any given such sequence a new number, *i.e.* a new element of $P(S)$.

Cantor further conjectured that $\aleph_1 = 2^{\aleph_0}$. That would mean that there is no infinite cardinality "between" that of $\mathbb{N}$ and that of $\mathbb{R}$. Cantor's conjecture is very famous under the name *Continuum Hypothesis* (CH). In the early 20^{th} century it was important enough for Hilbert to include it into his equally famous 23 problems that he assumed to be the most important math problems to be solved. It was, in fact, the first of these 23 problems.

The hypothesis as such is still unsolved today. In 1940, however, Kurt Gödel showed that CH cannot be disproven based on the standard axiomatic system, the Zermelo-Fraenkel set theory (ZF). In 1963, again, Paul Cohen showed that it cannot be proven in ZF either. Mathematicians today say that CH is *independent* from ZF. This may perhaps be translated into sloppy common speech as CH is irrelevant, at least in the context of the standard axiomatic system.

Without out too much phantasy, we can go beyond CH and suspect that there is a general rule that for any $n \in \mathbb{N} : \aleph_n = 2^{\aleph_{n-1}}$. $\aleph_1 = 2^{\aleph_0}$ would then be no exception. It would just be the way to count in infinity. This hypothesis is called the *Generalised Continuum Hypothesis* (GCH). The idea is extremely beautiful and, again, it makes a lot of sense. We cannot, of course, extend infinity by just adding an element; that would still be infinity. Between a set and its powerset, however, there is a *structural* difference that still holds, as we have seen, with infinite sets. There is no one-to-one mapping from the set to its powerset or, more formally worded, "there is no *surjection* from a set to its powerset". This is known as *Cantor's theorem* and Cantor's diagonal argument demonstrates that the relation between the sets $\mathbb{N}$ and $\mathbb{R}$ is an instance of this theorem.

The GCH, just as the weaker CH, is also independent from ZF. There are, however, some stronger implications when assuming the truth of GCH, as shown, again, by Kurt Gödel and by Polish mathematician Wacław Sierpiński (1882 – 1969). That, however, would lead us deeply into mathematical logic, which is not our topic here. More interesting appears to be the question what actually causes the difference between the cardinalities of $\mathbb{N}$ and $\mathbb{R}$.

This question is discussed in Cantor's first article on set theory, a tremendously important document in the history of math. In this short article – it has less than five pages – Cantor first proves that the set of real *algebraic* numbers is countable and then that the set of real numbers is not countable providing this way a new proof for the existence of the *transcendental* numbers and demonstrating that it is the transcendental numbers that make the set of real number uncountable.

To enumerate the algebraic numbers, Cantor uses a sophisticated trick, that involves mathematical machinery that is not yet at our disposal. He uses *polynomials* – those beasts will be a major topic of the next part – that are *irreducible* over $\mathbb{Q}$. He orders these polynomials and interprets algebraic numbers as the *roots*, *i.e.* the solutions, of the polynomials. The first ten polynomials and their roots are:

Polynomial	Root
x	0
$x+1$	-1
$x-1$	1
$x+2$	-2
$2x+1$	$-\frac{1}{2}$
$2x-1$	$\frac{1}{2}$
$x-2$	2
$x+3$	-3
x^2+x-1	$\frac{-1-\sqrt{5}}{2}$
x^2-2	$-\sqrt{2}$
$\dots$	$\dots$

The column "Root" contains the components of the enumeration of the algebraic numbers, which, hence, goes like

$$0, -1, 1, -2, -\frac{1}{2}, \frac{1}{2}, 2, -3, \frac{-1-\sqrt{5}}{2}, -\sqrt{2}, \dots$$

The enumeration technique is better than the one he used to enumerate the rationals, since it contains each algebraic number only once. Perhaps you remember that, using Cantor's original technique for enumerating the rationals, we had to filter out duplicates.

We see further that the sequence enumerates an extension of $\mathbb{Q}$ containing roots of rational numbers and more complex formulas including such roots. Since Cantor used the property of a real number being algebraic, *i.e.* appearing as a solution of a polynomial with rational coefficients, he guarantees that the resulting enumeration contains exactly all algebraic numbers. He, thus, proves that the cardinality of the set of algebraic numbers equals $|\mathbb{N}| = \aleph_0$. Since, as we have already seen, the set of real numbers has not the cardinality $\aleph_0$, this means that it is indeed the transcendental numbers that make $\mathbb{R}$ uncountable.

7.12. Review of the Number Zoo

We have, in the previous chapters, studied properties of numbers and problems that are related to things that are countable. On the way, we repeatedly met concepts related to sets with operations that show certain properties, in particular closure, associativity, identity and invertibility. We found such structures not only among numbers, but also in relation with other objects like strings and permutations.

We called a structure of the form

$$(S, \circ)$$

consisting of a set S and an operation $\circ$ a magma, if the operation is closed over S, *i.e.*

$$a, b \in S \to a \circ b \in S.$$

A magma is a universal structure found in many contexts, not only numbers. But we saw that the natural numbers, $\mathbb{N}$, form a magma together with both, addition and multiplication. We can even go further and predict that all number types that we have constructed by extending the notion of *natural number*, namely $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$, are magmas: they are all closed under the operations $+$ and $\times$.

But there are other properties, seen with $+$ and $\times$ together with any of our number types, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$. First associativity:

$$(a \circ b) \circ c = a \circ (b \circ c) = a \circ b \circ c.$$

This property makes all our number types semigroups. Furthermore, they all have an identity, e, together with either of the operations, namely 0 for addition and 1 for multiplication, such that for any $a \in S$:

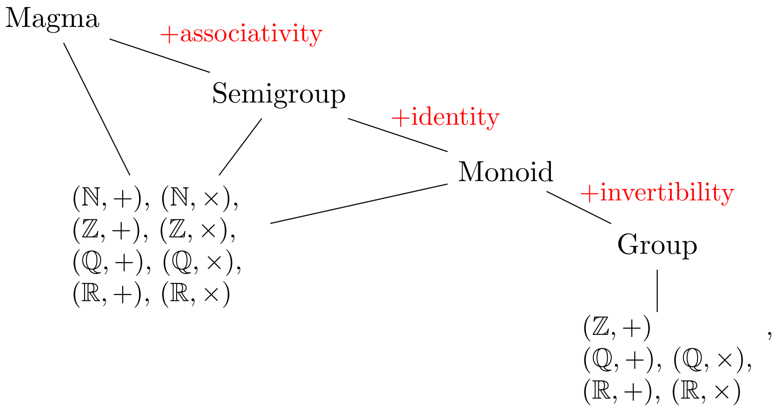
$$a \circ e = e \circ a = a.$$

This property makes all the number types together with either of the operations monoids.

Now, we have seen that some of the number types and operations, but not all of them, have yet another property, namely invertibility, *i.e.* the property that, for any $a \in S$, there is an element $a' \in S$, such that

$$a \circ a' = e.$$

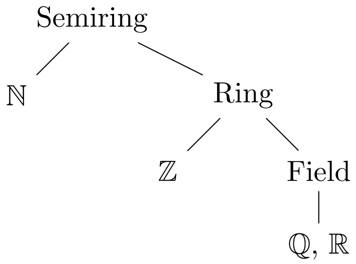
This property holds for $(\mathbb{Z}, +)$, $(\mathbb{Q}, +)$, $(\mathbb{Q}, \times)$ and $(\mathbb{R}, +)$ as well as $(\mathbb{R}, \times)$. Invertibility makes these structures groups. The following sketch summarises this result:


Magma
+associativity
Semigroup
+identity
Monoid
+invertibility
Group
$(\mathbb{N}, +)$, $(\mathbb{N}, \times)$,
$(\mathbb{Z}, +)$, $(\mathbb{Z}, \times)$,
$(\mathbb{Q}, +)$, $(\mathbb{Q}, \times)$,
$(\mathbb{R}, +)$, $(\mathbb{R}, \times)$
$(\mathbb{Z}, +)$,
$(\mathbb{Q}, +)$, $(\mathbb{Q}, \times)$,
$(\mathbb{R}, +)$, $(\mathbb{R}, \times)$

On top of these definitions, a different kind of structures is defined, that serves to distinguish different types of numbers. These new structures consist of a set and two operations, called addition and multiplication, respectively:

$$(S, +, \times).$$

If both operations form monoids over S, then we call this structure a **semiring**. An example of a semiring is $(\mathbb{N}, +, \times)$, since this structure consists of two monoids. If addition forms a group over S and multiplication forms a monoid, then we call this structure a **ring**. An example of a ring is $(\mathbb{Z}, +, \times)$, because addition in this structure is a group and multiplication is a monoid. If both, addition and multiplication, form a group over S, we call the resulting strucure a **field**. Examples for fields are $(\mathbb{Q}, +, \times)$ and $(\mathbb{R}, +, \times)$, since these structures have groups for both addition and multiplication. Here is an overview over our number types:


Semiring
$\mathbb{N}$
Ring
$\mathbb{Z}$
Field
$\mathbb{Q}$, $\mathbb{R}$

Part II.

Algebra and Geometry

8. Polynomials

8.1. Numeral Systems

A numeral system consists of a finite set of digits, D, and a base, b, for which $b = |D|$, *i.e.* b is the cardinality of D. The binary system, for instance, uses the digits $D = \{0, 1\}$. The cardinality of D in this case, hence, is 2. The decimal system uses the digits $D = \{0 \dots 9\}$ and, thus, has the base $b = 10$. The hexadecimal system uses the digits $D = \{0 \dots 15\}$, often given as $D = \{0 \dots 9, a, b, c, d, e, f\}$, and, therefore, has the base $b = 16$.

Numbers in any numeral system are usually represented as strings of digits. The string

$$10101010,$$

for instance, may represent a number in the binary system. (It could be a number in decimal format, too, though.) The string

$$170,$$

by contrast, cannot be a binary number, because it contains the digit 7, which is not element of D in the binary system. It can represent a decimal (or a hexadecimal number). The string

$$aa,$$

can represent a number in the hexadecimal system (but not one of in the binary or decimal system).

We interpret such a string, *i.e.* convert it to the decimal system, by rewriting it as a formula of the form:

$$a_n b^n + a_{n-1} b^{n-1} + \dots + a_0 b^0,$$

where a_i are the digits that appear in the string, b is the base and n is position of the left-most digit starting to count with 0 on the right-hand side of the string. The string 10101010 in binary notation,hence, is interpreted as

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0,$$

which can be simplified to

$$2^7 + 2^5 + 2^3 + 2,$$

which, in its turn, is

$$128 + 32 + 8 + 2 = 170.$$

The string 170 in decimal notation is interpreted as

$$10^2 + 7 \times 10 = 170.$$

Interpreting a string in the notation it is written in yields just that string. The string aa in hexadecimal notation is interpreted as

$$a \times 16 + a.$$

The digit a, however, is just 10. We, hence, get the equation

$$10 \times 16 + 10 = 160 + 10 = 170.$$

What do we get, when we relax some of the constraints defining a numeral system? Instead of using a finite set of digits, we could use a number field, F, (finite or infinite) so that any member of that field qualifies as coefficient in the formulas we used above to interpret numbers in the decimal system. We would then relax the rule that the base must be the cardinality of the field. Instead, we allow any member x of the field to serve as a base. Formulas we get from those new rules would follow the recipe:

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 x^0$$

or shorter:

$$\sum_{i=0}^{n} a_i x^i$$

with $a_i, x \in F$.

Such beasts are indeed well-known. They are very prominent, in fact, and their name is *polynomial*.

The name *poly*nomial stems from the fact that polynomials may be composed of many terms; a monomial, by contrast, is a polynomial that consists of only one term. For instance,

$$5x^2$$

is a monomial. A binomial is a polynomial that consists of two terms. This is an example of a binomial:

$$x^5 + 2x.$$

There is nothing special about monomials and binomials, at least nothing that would affect their definition as polynomials. Monomials and binomials are just polynomials that happen to have only one or, respectively, two terms.

Polynomials share many properties with numbers. Like numbers, arithmetic, including addition, subtraction, multiplication and division as well as exponentiation, can be defined over polynomials. In some cases, numbers reveal their close relation to polynomials. The binomial theorem states, for instance, that a product of the form

$$(a + b)(a + b)$$

translates to a formula involving binomial coefficients:

$$a^2 + 2ab + b^2.$$

We can interpret this formula as the product of the polynomial $x + a$:

$$(x + a)(x + a),$$

which yields just another polynomial:

$$x^2 + 2ax + a^2$$

Let us replace a for the number 3 and fix $x = 10$. We get:

$$(10 + 3)(10 + 3) = 10^2 + 2 \times 3 \times 10 + 3^2 = 100 + 60 + 9 = 169, \tag{8.1}$$

which is just the result of the multiplication 13 × 13. Usually, it is harder to recognise this kind of relations numbers have with the binomial theorem (and, hence, with polynomials), because most binomial coefficients are too big to be represented by a single-digit number. Already in the product 14 × 14, the binomial coefficients are hidden. Let us look at this multiplication treated as the polynomial $(x + a)$ with $x = 10$ and $a = 4$:

$$(10 + 4)(10 + 4) = 10^2 + 2 \times 4 \times 10 + 4^2 = 100 + 2 \times 40 + 16.$$

When we look at the resulting number, we do not recognise the binomial coefficient anymore – they are *carried* away: $100 + 2 \times 40 + 16 = 100 + 80 + 16 = 196$.

Indeed, polynomials are not numbers. Those are different concepts. Another important difference is that polynomials do not establish a clear order. For any two distinct numbers, we can clearly say which of the two is the greater and which is the smaller one. We cannot decide that based on the formula of the polynomial alone. One way to decide quickly which of two numbers is the grater one is to look at the number of their digits. The one with more digits is necessarily the greater one. In any numeral system it holds that:

$$a_3b^3 + a_2b^2 + a_1b + a_0 > c_2b^2 + c_1b + c_0$$

independent of the values of the as and the cs. For polynomials, this is not true. Consider the following example:

$$x^3 + x^2 + x + 1 > 100x^2?$$

For $x = 10$, the left-hand side of the inequation is $1000 + 100 + 10 + 1 = 1111$; the right-hand side, however, is $100 \times 100 = 10000$.

In spite of such differences, we can represent polynomials very similar to how we represented numbers, namely as a list of coefficients. This is a valid implementation in Haskell:

```haskell
type Poly a = P [a]
  deriving (Show)
```

We add a safe constructor:

```
poly :: (Eq a, Num a) ⇒ [a] → Poly a
poly [] = error "not a polynomial"
poly as = P (cleanz as)

cleanz :: (Eq a, Num a) ⇒ [a] → [a]
cleanz xs = reverse $ go (reverse xs)
  where go []       = []
        go [0]      = [0]
        go (0 : xs) = go xs
        go xs       = xs
```

The constructor makes sure that the resulting polynomial has at least one coefficient and that all the coefficients are actually numbers and comparable for equality. The function *cleanz* called in the constructor removes leading zeros (which are redundant), just as we did when we defined natural numbers. But note that we reverse, first, the list of coefficients passed to *go* and, second, the result of *go*. This means that we store the coefficients from left to right in ascending order. Usually, we write polynomials out in descending order of their weight, *i.e.*

$$x^n + x^{n-1} + \cdots + x^0.$$

But, here, we store them in the order:

$$x^0 + x^1 + \cdots + x^{n-1} + x^n.$$

The following function gets the list of coefficients back:

```
coeffs :: Poly a → [a]
coeffs (P as) = as
```

Here is a function to pretty-print polynomials:

```
pretty :: (Num a, Show a, Eq a) ⇒ Poly a → String
pretty p = go (reverse $ weigh p)
  where go [] = ""
        go ((i, c) : cs) = let x | i ≡ 0     = ""
                                 | i ≡ 1     = "x"
                                 | otherwise = "x^" ++ show i
                               t | c ≡ 1     = x
                                 | otherwise = show c ++ x
                               o | null cs   = ""
                                 | otherwise = " + "
                           in if c ≡ 0 then go cs else t ++ o ++ go cs
weigh :: (Num a) ⇒ Poly a → [(Integer, a)]
weigh (P []) = []
weigh (P as) = (zip [0 ..] as)
```

The function demonstrates how we actually interpret the list of coefficients. We first *weigh* them by zipping the list of coefficients with a list of integers starting at 0. One could say: we count the coefficients. Note that we start with 0, so that the first coefficient gets the weight 0, the second gets the weight 1 and so on. That, again, reflects our descending ordering of coefficients.

The reversed weighted list is then passed to *go*, which does the actual printing. We first determine the substring describing x: if i, the weight, is 0, we do not want to write the x, since $x^0 = 1$. If $i = 1$, we just write x. Otherwise we write x^i.

Then we determine the term composed of coefficient and x. If the coefficient, c is 1, we just write x; otherwise, we concatenate c with x. Note, however, that we later consider an additional case, namely, when $c = 0$. In this case, we ignore the whole term.

We still consider the operation. If the remainder of the list is *null*, *i.e.* we are now handling the last term, o is the empty string. Otherwise, it is the plus symbol. Here is room for improvement: when the coefficient is negative, we do not really need the operation, since we then write $+ - cx$. Nicer would be to write only $-cx$.

Finally, we put everything together concatenating a string composed of term, operation and *go* applied on the remainder of the list.

Here is a list of polynomials and how they are represented with our Haksell type:

$x^2 + x + 1$	*poly* $[1, 1, 1]$
$5x^5 + 4x^4 + 3x^3 + 2x^2 + x$	*poly* $[0, 1, 2, 3, 4, 5]$
$5x^4 + 4x^3 + 3x^2 + 2x + 1$	*poly* $[1, 2, 3, 4, 5]$
$5x^4 + 3x^2 + 1$	*poly* $[1, 0, 3, 0, 5]$

An important concept related to polynomials is the *degree*. The degree is a measurement of the *size* of the polynomial. In concrete terms, it is the greatest exponent in the polynomial. For us, it is the weight of the right-most element in the polynomial or, much simpler, the length of the list of coefficients minus one – since, we start with zero! The following function computes the degree of a given polynomial:

```
degree :: Poly a → Int
degree (P as) = length as − 1
```

Note, by the way, that polynomials of degree 0, those with only one trivial term, are just constant numbers.

Finally, here is a useful function that creates random polynomials with *Natural* coefficients:

```
randomPoly :: Natural → Int → IO (Poly Natural)
randomPoly n d = do
  cs ← cleanz < $ > mapM (\_ → randomCoeff n) [1 . . d]
  if length cs < d then randomPoly n d
    else return (P cs)
randomCoeff :: Natural → IO Natural
randomCoeff n = randomNatural (0, n − 1)
```

The function receives a *Natural* and an *Int*. The *Int* indicates the degree of the polynomial we want to obtain. The *Natural* is used to restrict the size of the coefficients we want to see in the polynomial. In *randomCoeff*, we use the *randomNatural* defined in the previous chapter to generate a random number between 0 and $n - 1$. You might suspect already where that will lead us: to polynomials modulo some number. But before we get there, we will study polynomial arithmetic.

8.2. Polynomial Arithmetic

We start with addition and subtraction, which, in German, are summarised by the beautiful word *strichrechnung* meaning literally "dash calculation" as opposed to *punktrechnung* or "dot calculation", which would be multiplication and division.

Polynomial *strichrechnung* is easy. Key is to realise that the structure of polynomials is already defined by *strichrechnung*: it is composed of terms each of which is a product of some number and a power of x. When we add (or subtract) two polynomials, we just sort them according to the exponents of their terms and add (or subtract) terms with equal exponents:

$$\begin{array}{cccccccccc} & ax^n & + & bx^{n-1} & + & \dots & + & c \\ + & dx^n & + & ex^{n-1} & + & \dots & + & f \\ = & (a+d)x^n & + & (b+e)x^{n-1} & + & \dots & + & c+f \end{array} \tag{8.2}$$

With our polynomial representation, it is easy to implement this kind of operation. One might think it was designed especially to support addition and subtraction. Here is a valid implementation:

```
add :: (Num a, Eq a) ⇒ Poly a → Poly a → Poly a
add = strich (+)

sub :: (Num a, Eq a) ⇒ Poly a → Poly a → Poly a
sub = strich (−)

sump :: (Num a, Eq a) ⇒ [Poly a] → Poly a
sump = foldl′ add (P [0])

strich :: (Num a, Eq a) ⇒ (a → a → a) → Poly a → Poly a → Poly a
strich o (P x) (P y) = P (strichlist o x y)

strichlist :: (Num a, Eq a) ⇒ (a → a → a) → [a] → [a] → [a]
strichlist o xs ys = cleanz (go xs ys)
  where go [] bs             = bs
        go as []             = as
        go (a : as) (b : bs) = a ‘o‘ b : go as bs
```

Note that *sump* function, which implements *sum* for polynomials. Here is one more function that might be useful later on; it folds *strichlist* on a list of lists of coefficients:

```
strichf :: (Num a, Eq a) ⇒ (a → a → a) → [[a]] → [a]
strichf o = foldl′ (strichlist o) []
```

Punktrechnung, *i.e.* multiplication and division, are a bit more complex – because of the distribution law. Let us start with the simple case where we distribute a monomial over a polynomial:

```
mul1 :: Num a ⇒ (a → a → a) → Int → [a] → a → [a]
mul1 o i as a = zeros i ++ go as a
  where go [] _       = []
        go (c : cs) x = c ‘o‘ x : go cs x

zeros :: Num a ⇒ Int → [a]
zeros i = take i $ repeat 0
```

The function *mul1* takes a single term (the monomial) and distributes it over the coefficients of a polynomial using the operation *o*. Each term in the polynomial is combined with the single term. This corresponds to the operation:

$$\begin{array}{ccccccccc} dx^m & \times & ax^n & + & bx^{n-1} & + & \dots & + & c \\ & = & adx^{m+n} & + & bdx^{n-1+m} & + & \dots & + & cdx^m \end{array} \tag{8.3}$$

The function *mul1* receives on more parameter, namely the *Int i* and uses it to generate a sequence of zeros that is put in front of the resulting coefficient list. As we will see shortly, the list of zeros reflects the weight of the single term. In fact, we do not implement the manipulation of the exponents we see in the abstract formula directly. Instead, the addition $+m$ is implicitly handled by placing m zeros at the head of the list resulting in a new polynomial of degree $m + d$ where d is the degree of the original polynomial. A simple example:

$$5x^2 \times (4x^3 + 3x^2 + 2x + 1) = 20x^5 + 15x^4 + 10x^3 + 5x^2$$

would be:

mul1 $2\ [1, 2, 3, 4]\ 5$

which is:

zero $2 +\!\!+ (5 * [1, 2, 3, 4]) = [0, 0, 5, 10, 15, 20]$

We, hence, would add 2 zeros, since 2 is the degree of the monomial.

Now, when we multiply two polynomials, we need to map all terms in one of the polynomials on the other polynomial using *mul1*. We further need to pass the weight of the individual terms of the first polynomial as the *Int* parameter of *mul1*. What we want to do is:

$[\mathit{mul1}\ (*)\ i\ (\mathit{coeffs}\ \mathit{p1})\ p \mid (i, p) \leftarrow \mathit{zip}\ [0\,..]\ (\mathit{coeffs}\ \mathit{p2})]$.

What would we get applying this formula on the polynomials, say, $[1, 2, 3, 4]$ and $[5, 6, 7, 8]$? Let us have a look:

$[\mathit{mul1}\ (*)\ i\ ([5, 6, 7, 8])\ p \mid (i, p) \leftarrow \mathit{zip}\ [0\,..]\ [1, 2, 3, 4]]$
$[[5, 6, 7, 8], [0, 10, 12, 14, 16], [0, 0, 15, 18, 21, 24], [0, 0, 0, 20, 24, 28, 32]]$.

We see a list of four lists, one for each coefficient of $[1, 2, 3, 4]$. The first list is the result of distributing 1 over all the coefficients in $[5, 6, 7, 8]$. Since 1 is the first element, its weight is 0: no zeros are put before the resulting list. The second list results from distributing 2 over $[5, 6, 7, 8]$. Since 2 is the second element, its weight is 1: we add one zero. The same process is repeated for 3 and 4 resulting in the third and fourth result list. Since 3 is the the third element, the third resulting list gets two zeros and, since 4 is the fourth element, the fourth list gets three zeros.

How do we transform this list of lists back into a single list of coefficients? Very easy: we add them together using *strichf*:

strichf $(+)\ [[5,6,7,8],[0,10,12,14,16],[0,0,15,18,21,24],[0,0,0,20,24,28,32]]$

which is

$[5,16,34,60,61,52,32]$.

This means that

$$(4x^3+3x^2+2x+1)\times(8x^3+7x^2+6x+5) = 32x^6+52x^5+61x^4+60x^3+34x^2+16x+5. \tag{8.4}$$

Here is the whole algorithm:

```
mul :: (Show a, Num a, Eq a) ⇒ Poly a → Poly a → Poly a
mul p1 p2 | d2 > d1   = mul p2 p1
          | otherwise = P (strichf (+) ms)
  where d1 = degree p1
        d2 = degree p2
        ms = [mul1 (∗) i (coeffs p1) p ∨ (i, p) ← zip [0 ..] (coeffs p2)]
```

On top of multiplication, we can implement power. We will, of course, not implement a naïve approach based on repeated multiplication alone. Instead, we will use the *square-and-multiply* approach we have already used before for numbers. Here is the code:

```
powp :: (Show a, Num a, Eq a) ⇒ Natural → Poly a → Poly a
powp f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mul y x
        go n y x | even n    = go (n ‘div‘ 2) y       (mul x x)
                 | otherwise = go ((n − 1) ‘div‘ 2) (mul y x)
                                                      (mul x x)
```

The function *powp* receives a natural number, that is the exponent, and a polynomial. We kick off by calling *go* with the exponent, f, a base polynomial P [1], *i.e.* unity, and the polynomial we want to raise to the power of f. If $f = 0$, we are done and return the base polynomial. This reflects the case $x^0 = 1$. If $f = 1$, we multiply the base polynomial by the input polynomial. Otherwise, if the exponent is even, we halve it, pass the base polynomial on and square the input. Otherwise, we pass the product of the base polynomial and the input on instead of the base polynomial as it is. This implementation differs a bit from the implementation we presented before for numbers, but it implements the same algorithm.

Here is a simple example: we raise the polynomial $x + 1$ to the power of 5. In the first round, we compute

go $5\ (P\ [1])\ (P\ [1,1])$,

which, since 5 is odd, results in

$go\ 2\ (P\ [1,1])\ (P\ [1,2,1])$.

This, in its turn, results in

$go\ 1\ (P\ [1,1])\ (P\ [1,4,6,4,1])$.

This is the final step and results in

$mul\ (P\ [1,1])\ (P\ [1,4,6,4,1])$,

which is

$P\ [1,5,10,10,5,1]$,

the polynomial $x^5+5x^4+10x^3+10x^2+5x+1$. You might have noticed that our Haskell notation shows the binomial coefficients $\binom{n}{k}$ for $n=0$, $n=1$, $n=2$, $n=4$ and $n=5$. We never see $n=3$, which would be $P\ [1,3,3,1]$, because we leave the multiplication $mul\ (P\ [1,1])\ (P\ [1,2,1])$ out. For this specific case with exponent 5, leaving out this step is where square-and-multiply is more efficient than multiplying five times. With growing exponents, the saving quickly grows to a significant order.

Division is, as usual, a bit more complicated than multiplication. But it is not too different from number division. First, we define polynomial division as Euclidean division, that is we search the solution for the equation

$$\frac{a}{b} = q + r \tag{8.5}$$

where $r < b$ and $bq + r = a$.

The manual process is as follows: we divide the first term of a by the first term of b. The quotient goes to the result; then we multiply it by b and set a to a minus that result. Now we repeat the process until the degree of a is less than that of b.

Here is an example:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2 + 1}.$$

We start by dividing $4x^5$ by x^2. The quotient is $4x^3$, which we add to the result. We multiply: $4x^3 \times (x^2+1) = 4x^5 + 4x^3$ and subtract the result from a:

$$\begin{array}{cccccccccc} & 4x^5 & - & x^4 & + & 2x^3 & + & x^2 & - & 1 \\ - & 4x^5 & & & + & 4x^3 & & & & \\ = & & - & x^4 & - & 2x^3 & + & x^2 & - & 1 \end{array} \tag{8.6}$$

We continue with $-x^4$ and divide it by x^2, which is $-x^2$. The overall result now is

$4x^3 - x^2$. We multiply $-x^2 \times (x^2+1) = -x^4 - x^2$ and subtract that from what remains from a:

$$\begin{array}{rrrrrrrrr} & - & x^4 & - & 2x^3 & + & x^2 & - & 1 \\ - & - & x^4 & & & - & x^2 & & \\ = & & & - & 2x^3 & + & 2x^2 & - & 1 \end{array} \tag{8.7}$$

We continue with $-2x^3$, which, divided by x^2 is $-2x$. We multiply $-2x \times (x^2+1) = -2x^3 - 2x$ and subtract:

$$\begin{array}{rrrrrrrrr} & - & 2x^3 & + & 2x^2 & + & & - & 1 \\ - & - & 2x^3 & & & - & 2x & & \\ = & & & & 2x^2 & + & 2x & - & 1 \end{array} \tag{8.8}$$

The result now is $4x^3 - x^2 - 2x$. We continue with $2x^2$, which, divided by x^2 is 2. We multiply $2 \times (x^2+1) = 2x^2 + 2$ and subtract:

$$\begin{array}{rrrrrr} & 2x^2 & + & 2x & - & 1 \\ - & 2x^2 & & & + & 2 \\ = & & & 2x & - & 3 \end{array} \tag{8.9}$$

The result now is $4x^3 - x^2 - 2x + 2$. We finally have $2x - 3$, which is smaller in degree than b. The result, hence, is $(4x^3 - x^2 - 2x + 2, 2x - 3)$.

Here is an implementation of division in Haskell:

```
divp ::    (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
           Poly a → Poly a → (Poly a, Poly a)
divp (P as) (P bs) = let (q, r) = go [] as in (P q, P r)
  where db = degree (P bs)
        go q r | degree (P r) < db = (q, r)
               | null r ∨ r ≡ [0]  = (q, r)
               | otherwise          =
                 let t  = last r / last bs
                     d  = degree (P r) − db
                     ts = zeros d ++ [t]
                     m  = mulist ts bs
                 in go (cleanz $ strichlist (+) q ts)
                       (cleanz $ strichlist (−) r m)
mulist :: (Show a, Num a, Eq a) ⇒ [a] → [a] → [a]
mulist c1 c2 = coeffs $ mul (P c1) (P c2)
```

First note that division expects its arguments to be polynomials over a *Fractional* data type. We do not allow polynomials over integers to be used with this implementation.

The reason is that we do not want to use Euclidean division on the coefficients. That could indeed be very confusing. Furthermore, polynomials are most often used with rational or real coefficients. Restricting division to integers (using Euclidean division) would, therefore, not make much sense.

Observe further that we call *go* with an empty set – that is the initial value of q, *i.e.* the final result – and *as* – that is initially the number to be divided, the number we called a above. The function *go* has two base cases: if the degree of r, the remainder and initially *as*, is less than the degree of the divisor b, we are done. The result is our current (q, r). The same is true if r is *null* or contains only the constant 0. In this case, there is no remainder: b divides a.

Otherwise, we divide the *last* of r by the *last* of b. Note that those are the term with the highest degree in each polynomial. This division is just a number division of the two coefficients. We still have to compute the new exponent, which is the exponent of *last* r minus the exponent of *last* b, *i.e.* their weight. We do this by subtracting their degrees and then inserting zeros at the head of the result *ts*. This result, *ts*, is then added to q. We further compute $ts \times bs$ and subtract the result from r. The function *mulist* we use for this purpose is just a wrapper around *mul* using lists of coefficients instead of *Poly* variables. With the resulting (q, r), we go into the next round.

Let us try this with our example from above:

$$\frac{4x^5 - x^4 + 2x^3 + x^2 - 1}{x^2 + 1}.$$

We call *divp* $(P\ [-1, 0, 1, 2, -1, 4])$ $(P\ [1, 0, 1])$ and get $(P\ [2, -2, -1, 4], P\ [-3, 2])$, which translates to the polynomials $4x^3 - x^2 - 2x + 2$ and $2x - 3$. This is the same result we obtained above with the manual procedure.

From here on, we can implement functions based on division, such as *divides*:

```
divides :: (Show a, Num a, Eq a, Ord a) ⇒
             Poly a → Poly a → Bool
divides a b = case b `divp` a of
                (_, P [0]) → True
                _          → False
```

the remainder:

```
remp :: (Show a, Num a, Eq a, Ord a) ⇒
          Poly a → Poly a → Bool
remp a b = let (_, r) = b `d` a in r
```

and, of course, the GCD:

```
gcdp :: (Show a, Num a, Eq a, Fractional a, Ord a) ⇒
        Poly a → Poly a → Poly a
gcdp a b | degree b > degree a = gcdp b a
         | zerop b     = a
         | otherwise = let (_, r) = divp a b in gcdp b r
```

We use a simple function to check whether a polynomial is zero:

```
zerop :: (Num a, Eq a) ⇒ Poly a → Bool
zerop (P [0]) = True
zerpo _       = False
```

We can demonstrate *gcdp* nicely on binomial coefficients. For instance, the GCD of the polynomials $x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$ and $x^3 + 3x^2 + 3x + 1$, thus

```
gcdp (P [1,5,10,10,5,1]) (P [1,3,3,1])
```

is $x^3 + 3x^2 + 3x + 1$.

Since polynomials consisting of binomial coefficients of n, where n is the degree of the polynomial, are always a product of polynomials composed of smaller binomial coefficients in the same way, the GCD of two polynomials consisting only of binomial coefficients, is always the smaller of the two. In other cases, that is, when the smaller does not divide the greater, this implementation of the GCD can lead to confusing results. For instance, we multiply $P\ [1,2,1]$ by another polynomial, say, $P\ [1,2,3]$. The result is $P\ [1,4,8,8,3]$. Now,

```
gcdp (P [1,5,10,10,5,1]) (P [1,4,8,8,3])
```

does not yield the expected result $P\ [1,2,1]$. The reason is that the GCD is an operation defined on integers, but we implemented it on top of fractionals. That is often not what we want. Anyway, here, we will actually use the GCD only in finite fields. Until now, we have discussed polynomials in infinite fields. We now turn our attention to polynomial arithmetic in a finite field and, hence, to modular polynomial arithmetic.

With modular arithmetic, all coefficients in the polynomial are modulo n. That means we have to reduce those numbers. This, of course, does only make sense with integers. We first implement some helpers to reduce numbers modulo n reusing functions implemented in the previous chapter.

The first function takes an integer modulo n:

```
mmod :: Zahl → Zahl → Zahl
mmod n p | n < 0 ∧ (−n) > p = mmod (−(mmod (−n)) p) p
         | n < 0             = mmod (p + n) p
         | otherwise         = n ‘rem‘ p
```

Equipped with this function, we can easily implement multiplication:

```
modmul :: Zahl → Zahl → Zahl → Zahl
modmul p f1 f2 = (f1 ∗ f2) ‘mmod‘ p
```

For division, we reuse the *inverse* function:

```
modiv :: Zahl → Zahl → Zahl → Zahl
modiv p n d = modmul p n d′
  where d′ = M.inverse d p
```

Now, we turn to polynomials. Here is, first, a function that transforms a polynomial into one modulo n:

```
pmod :: Poly Zahl → Zahl → Poly Zahl
pmod (P cs) p = P [c ‘mmod‘ p | c ← cs]
```

In other words, we just map *mmod* on all coefficients. Let us look at some polynomials modulo a number, say, 7. The polynomial `P [1, 2, 3, 4]` we already used above is just the same modulo 7. The polynomial `P [5, 6, 7, 8]`, however, changes:

```
P [5, 6, 7, 8] ‘pmod‘ 7
```

is `P [5, 6, 0, 1]` or, in other words, $8x^3 + 7x^2 + 6x + 5$ turns, modulo 7, into $x^3 + 6x + 5$.

The polynomial $x+1$ raised to the power of 5 is $x^5+5x^4+10x^3+10x^2+5x+1$. Modulo 7, this reduces to $x^5+5x^4+3x^3+3x^3+5x+1$. That is: the binomial coefficients modulo n change. For instance,

```
map (choose2 6) [0 . . 6]
```

is

1,6,15,20,15,6,1.

Modulo 7, we get

1,6,1,6,1,6,1.

```
map (choose2 7) [0 . . 7]
```

is

1,7,21,35,35,21,7,1.

Without big surprise, we see this modulo 7 drastically simplified:

1,0,0,0,0,0,0,1.

Here are addition and subtraction, which are very easy to convert to modular arithmetic:

```
addmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
addmp n p1 p2 = strich (+) p1 p2 ‘pmod‘ n
submp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
submp n p1 p2 = strich (−) p1 p2 ‘pmod‘ n
```

Multiplication:

```
mulmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
mulmp p p1 p2 | d2 > d1   = mulmp p p2 p1
              | otherwise = P [m ‘mmod‘ p | m ← strichf (+) ms]
  where ms = [mul1 o i (coeffs p1) c | (i, c) ← zip [0 ..] (coeffs p2)]
        d1 = degree p1
        d2 = degree p2
        o  = modmul p
```

We repeat the multiplication from above

`mul (P [1, 2, 3, 4]) (P [5, 6, 7, 8])`

which was

`P [5, 16, 34, 60, 61, 52, 32]`

Modulo 7, this result is

`P [5, 2, 6, 4, 5, 3, 4]`.

The modulo multiplication

`mulmp 7 (P [1, 2, 3, 4]) (P [5, 6, 0, 1])`

yields the same result:

`P [5, 2, 6, 4, 5, 3, 4]`

Division:

```
divmp :: Zahl → Poly Zahl → Poly Zahl → (Poly Zahl, Poly Zahl)
divmp p (P as) (P bs) = let (q, r) = go [0] as in (P q, P r)
  where db = degree (P bs)
        go q r | degree (P r) < db = (q, r)
               | null r ∨ r ≡ [0]  = (q, r)
               | otherwise         =
                 let t  = modiv p (last r) (last bs)
                     d  = degree (P r) − db
                     ts = zeros d ++ [t]
                     m  = mulmlist p ts bs
                 in go [c ‘mmod‘ p | c ← cleanz $ strichlist (+) q ts]
                       [c ‘mmod‘ p | c ← cleanz $ strichlist (−) r m]
```

GCD:

```
gcdmp :: Zahl → Poly Zahl → Poly Zahl → Poly Zahl
gcdmp p a b | degree b > degree a = gcdmp p b a
            | zerop b = a
            | otherwise = let (_, r) = divmp p a b in gcdmp p b r
```

Let us try *gcdmp* on the variation we already tested above. We multiply the polynomial $x^2 + 2x + 1$ by $3x^2 + 2x + 1$ modulo 7:

mulmp 7 (*P* [1, 2, 1]) (*P* [1, 2, 3]).

The result is *P* [1, 4, 1, 1, 3].

Now, we compute the GCD with *P* [1, 5, 10, 10, 5, 1] modulo 7:

gcdmp 7 (*P* [1, 5, 3, 3, 5, 1]) (*P* [1, 4, 1, 1, 3]).

The result is *P* [1, 2, 1], as expected.

Finally, power:

```
powmp :: Zahl → Zahl → Poly Zahl → Poly Zahl
powmp p f poly = go f (P [1]) poly
  where go 0 y _ = y
        go 1 y x = mulmp p y x
        go n y x | even n    = go (n ‘div‘ 2) y       (mulmp p x x)
                 | otherwise = go ((n − 1) ‘div‘ 2) (mulmp p y x)
                                                    (mulmp p x x)
```

Here is a nice variant of Pascal's triangle generated by *map* ($\lambda x \rightarrow$ *powmp* 7 x (*P* [1, 1]) [1 . . 14]:

```
                   P [1,1]
                  P [1,2,1]
                 P [1,3,3,1]
                P [1,4,6,4,1]
               P [1,5,3,3,5,1]
              P [1,6,1,6,1,6,1]
             P [1,0,0,0,0,0,0,1]
            P [1,1,0,0,0,0,0,1,1]
           P [1,2,1,0,0,0,0,1,2,1]
          P [1,3,3,1,0,0,0,1,3,3,1]
         P [1,4,6,4,1,0,0,1,4,6,4,1]
        P [1,5,3,3,5,1,0,1,5,3,3,5,1]
       P [1,6,1,6,1,6,1,1,6,1,6,1,6,1]
      P [1,0,0,0,0,0,0,2,0,0,0,0,0,0,1]
```

It is especially interesting to look at greater powers using exponents that are multiples of 7. Before we continue with modular arithmetic, which we need indeed to understand some of the deeper problems related to polynomials, we will investigate the application of polynomials using a famous device: Babbage's difference engine.

8.3. The Difference Engine

Polynomial arithmetic, as we have seen, is very similar to number arithmetic. What is the correspondent of interpreting a number in a given numeral system in the domain of polynomials? Well, that is the *application* of the polynomial to a given number. We would substitute x for a number in the Field in which we are working and just compute the formula. For instance, the polynomial

$$x^2 + x + 1$$

can be applied to, say, 2. Then we get the formula

$$2^2 + 2 + 1,$$

which is $4 + 2 + 1 = 7$.

For other values of x, it would of course generate other values. For $x = 0$, for instance, it would give $0^2+0+1 = 1$; for $x = 1$, it is $1^2+1+1 = 3$; for $x = 3$, it yields $3^2+3+1 = 13$.

How would we apply a polynomial represented by our Haskell type? We would need to go through the list of coefficients, raise x to the power of the weight of each particular coefficient, multiply it by the coefficient and, finally, add all the values together. Here is an implementation:

```
apply :: Num a ⇒ Poly a → a → a
apply (P cs) x = sum [c ∗ x ↑ i | (i, c) ← zip [0 ..] cs]
```

Let us try with a very simple polynomial, $x + 1$:

apply (P [1, 1]) 0 gives 1.
apply (P [1, 1]) 1 gives 2.
apply (P [1, 1]) 2 gives 3.
apply (P [1, 1]) 3 gives 4.

This polynomial, apparently, just counts the integers adding one to the value to which we apply it. It implements `i++`.

On the first sight, this result appears to be boring. However, after a quick thought, there is a lesson to learn: we get to know the polynomial, when we look at the *sequence* it produces. So, let us implement a function that maps *apply* to lists of numbers:

```
mapply :: Num a ⇒ Poly a → [a] → [a]
mapply p = map (apply p)
```

For simple polynomials, the sequences are predictable. x^2, obviously, just produces the squares; x^3 produces the cubes and so on. Sequences created by powers of the simple polynomial $x + 1$, still, are quite predictable, *e.g.*

mapply (P $[1, 2, 1]$) $[0\,..\,10]$: 1,4,9,16,25,36,49,64,81,100,121
mapply (P $[1, 3, 3, 1]$) $[0\,..\,10]$: 1,8,27,64,125,216,343,512,729,1000,1331
mapply (P $[1, 4, 6, 4, 1]$) $[0\,..\,10]$: 1,16,81,256,625,1296,2401,4096,6561,10000,14641
mapply (P $[1, 5, 10, 10, 5, 1]$) $[0\,..\,10]$:
1,32,243,1024,3125,7776,16807,32768,59049,100000,161051

The first line, easy to recognise, is the squares, but pushed one up, *i.e.* the application to 0 yields the value for 1^2, the application to 1 yields the value for 2^2 and so on. The second, still easy to recognise, is the cubes – again pushed up by one. The third line is the powers of four and the fourth line is the powers of five, both pushed up by one.

That is not too surprising at the end, since P $[1, 2, 1]$ is the result of squaring P $[1, 1]$, which generates the integers pushed one up; P $[1, 3, 3, 1]$ is the result of raising P $[1, 1]$ to the third power and so on.

Things become more interesting, when we deviate from binomial coefficients. The sequence produced by *mappy* (P $[1, 2, 3, 4]$) $[1\,..\,10]$, for instance, does not resemble such a simple sequence: 1, 10, 49, 142, 313, 586, 985, 1534, 2257, 3178, 4321. Even the Online Encyclopedia has nothing interesting to say about it. The same is true for *mappy* (P $[5, 6, 7, 8]$) $[1\,..\,10]$, which is 5, 26, 109, 302, 653, 1210, 2021, 3134, 4597, 6458, 8765.

This raises another interesting question: given a sequence, is there a method by which we can we recognise the polynomial that created it? Yes, there is. In fact, there are. There was even a machine that helped guessing polynomials from sequences. It was built in the early 19^{th} century by Charles Babbage (1791 – 1871), an English polymath, mathematician, philosopher, economist and inventor.

Babbage stands in the tradition of designers and constructors of early computing machinery; predecessors of his in this tradition were, for instance, Blaise Pascal (1623 – 1662) and Gottfried Wilhelm Leibniz (1646 – 1716). Babbage designed two series of machines, first, the difference engines and, later, the analytical engines.

The analytical engine, unfortunately, was not built in his lifetime. The final collapse of the project came in 1878, after Babbage's death in 1871, due to lack of finance. The analytical engine would have been a universal (Turing-complete) computer very similar to our computers today, but not working on electricity, but on steam and brawn. It would have been programmed by punch cards that, in Babbage's time, were used for controlling looms. Programs would have resembled modern assembly languages allowing control structures like selection and iteration. In the context of a description of the analytical engine, Ada Lovelace (1815 – 1852), a friend of Babbage and daughter of Lord Byron, described how to compute Bernoulli numbers with the machine. She is, therefore, considered the first computer programmer in history.

The difference engine, at which we will look here, is much simpler. It was designed to analyse polynomials and what it did was, according to Babbage, "computing differences". During Babbage's lifetime, a first version was built and sucessfully demonstrated. The construction of a second, much more powerful version which was financially backed by the government, failed due to disputes between Babbage and his engineers. This machine was finally built by the London Science Museum in 1991 using material and engineering techniques available in the 19^{th} century proving this way that it was actually possible for Babbage and his engineers to build such a machine.

The difference engine, as Babbage put it, computes differences, namely the differences in a sequence of numbers. It would take as input a sequence of the form

0,1,16,81,256,625,1296,2401,4096,6561,10000

and compute the differences between the single numbers:

$$\begin{array}{rcrcr} 1 & - & 0 & = & 1 \\ 16 & - & 1 & = & 15 \\ 81 & - & 16 & = & 65 \\ 256 & - & 81 & = & 175 \\ \ldots & & & & \end{array} \tag{8.10}$$

Here is a simple function that does this job for us:

$$\begin{array}{l} \mathit{diffs} :: [\mathit{Zahl}] \rightarrow [\mathit{Zahl}] \\ \mathit{diffs}\ [\,] \qquad\qquad = [\,] \\ \mathit{diffs}\ [\_] \qquad\quad\ = [\,] \\ \mathit{diffs}\ (a : b : cs) = (b - a) : \mathit{diffs}\ (b : cs) \end{array}$$

Applied on the sequence above, *diffs* yields:

1,15,65,175,369,671,1105,1695,2465,3439

What is so special about it? Perhaps, nothing. But let us repeat the process using this sequence. It yields:

14,50,110,194,302,434,590,770,974

And once again:

36,60,84,108,132,156,180,204

And one more time:

24,24,24,24,24,24,24

Suddenly, we have a constant list. How often did we apply *diffs*? Four times – and, as you may have realised, the original sequence was generated by the polynomial x^4, a polynomial of degree 4. Is that coincidence?

For further investigation, we implement the complete difference machine, which takes differences, until it reaches a constant sequence.

```
dengine :: [Zahl] → [[Zahl]]
dengine cs | constant cs = [ ]
           | otherwise = ds : dengine ds
  where ds = diffs cs
        constant [ ]      = True
        constant [_]      = True
        constant (x : xs) = all (≡ x) xs
```

Note that we restrict coefficients to integers. This is just for clarity. Usually, polynomials are defined over a field, such as the rational or the real numbers.

To confirm our suspicion that the difference engine creates n difference sequences for a polynomial of degree n, we apply the engine on x, x^2, x^3, x^4 and x^5 and count the sequences it creates:

length (*dengine* (*mapply* (*P* [0, 1]) [0 . . 32])): 1
length (*dengine* (*mapply* (*P* [0, 0, 1]) [0 . . 32])): 2
length (*dengine* (*mapply* (*P* [0, 0, 0, 1]) [0 . . 32])): 3
length (*dengine* (*mapply* (*P* [0, 0, 0, 0, 1]) [0 . . 32])): 4
length (*dengine* (*mapply* (*P* [0, 0, 0, 0, 0, 1]) [0 . . 32])): 5

The engine already has a purpose: it tells us the degree of the polynomial that generates a given sequence. It can do much more, though. For instance, it lets us predict the next value in the sequence. To do so, we take the constant difference from the last sequence and add it to the last difference of the previous sequence; we take that result and add it to the previous sequence and so on, until we reach the first sequence. Consider the sequence and its differences from above:

0,1,16,81,256,625,1296,2401,4096,6561,10000
1,15,65,175,369,671,1105,1695,2465,3439
14,50,110,194,302,434,590,770,974
36,60,84,108,132,156,180,204
24,24,24,24,24,24,24

We start at the bottom and compute $204 + 24 = 228$. This is the next difference of the previous sequence. We compute $974 + 228 = 1202$. We go one line up and compute $3439 + 1202 = 4641$. This, finally, is the difference to the next value in the input sequence, which, hence, is $10000 + 4641 = 14641$ and, indeed, 11^4. Even without knowing the polynomial that actually generates the sequence, we are now able to continue this sequence. Here is a function that does that for us:

```
predict :: [[Zahl]] → [Zahl] → Maybe Zahl
predict ds xs = case go (reverse ds) of
                  0 → Nothing
                  d → Just (d + (last xs))
   where go [] = 0
            go (a : cs) = last a + go cs
```

The function takes two arguments: the first is the list of difference sequences and the second is the original sequence. We apply *go* on the reverse of the sequences (because we are working backwards). For each sequence in this list, we get the last and add it to the last of the previous until we have exhausted the list. If *go* yields 0, we assume that something went wrong. The list of sequences may have been empty in the first place. Otherwise, we add the result to the last of the original list.

Here are some more examples:

let *s* = *mapply* (*P* [0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 11
let *s* = *mapply* (*P* [0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 121
let *s* = *mapply* (*P* [0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 1331
let *s* = *mapply* (*P* [0, 0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 14641
let *s* = *mapply* (*P* [0, 0, 0, 0, 0, 1]) [0 . . 10] **in** *predict* (*dengine s*) *s*: 161051

Let us go back to the question of how to find the polynomial given the sequence that this polynomial generates. With the help of the difference engine, we already know the degree of the polynomial. Supposed, we know that the first element in the sequence was generated applying 0 to the unknown polynomial and the second one was generated applying 1, the third by applying 2 and so on, we have all information we need.

From the degree, we know the form of the polynomial. A polynomial of degree 1 has the form $a_1x + a_2$; a polynomial of degree 2 has the form $a_1x^2 + a_2x + a_3$; a polynomial of degree 3 has the form $a_1x^3 + a_2x^2 + a_3x + a_4$ and so on.

Since we know the values to which the polynomial is applied, we can easily compute the value of the x-part of the terms. They are that value raised to the power of the weight. The challenge, then, is to find the coefficient by which that value is multiplied.

The first element in the sequence, the one created by applying the polynomial to 0, is just the last coefficient, the one "without" an x, since the other terms "disappear", when we apply to 0. Consider for example a polynomial of the form $x^2 + x + a$. When we apply it to 0, we get $0^2 + 0 + a = c$, where c is the first value in the sequence. Thus, $a = c$.

The second element is 1 applied to the formula and, therefore, all terms equal their coefficients, since cx^n, for $x = 1$, is just c. The third element results from applying 2 to the polynomial, it hence adheres to a formula where unknown values (the coefficients) are multiplied by 2, $2^2 = 4$, $2^3 = 8$ and so on.

In other words, for a polynomial of degree n, we can devise a system of linear equations with $n + 1$ unknowns and the $n + 1$ first elements of the sequence as constant values. A polynomial of degree 2, for instance, yields the system

$$\begin{array}{ccccccc} & & & & a & = & a_1 \\ a & + & b & + & c & = & a_2 \\ a & + & 2b & + & 4c & = & a_3 \end{array} \tag{8.11}$$

where the constant numbers a_1, a_2 and a_3 are the first three elements of the sequence. A polynomial of degree 3 would generate the system

$$\begin{array}{ccccccccc} & & & & & & a & = & a_1 \\ a & + & b & + & c & + & d & = & a_2 \\ a & + & 2b & + & 4c & + & 8d & = & a_3 \\ a & + & 3b & + & 9c & + & 27d & = & a_4 \end{array} \tag{8.12}$$

We have already learnt how to solve such systems: we can apply Gaussian elimination. The result of the elminiation is the coefficients of the generating polynomial, which are the unknowns in the linear equations. The known values (which we would call the coefficients in a linear equation) are the values obtained by computing x^i where i is the weight of the coefficient. Here is a function to extract the known values, the $x$es raised to the weight, from a given sequence with a given degree:

```
genCoeff :: Zahl → Zahl → Zahl → [Zahl]
genCoeff d n x = go 0 x
  where go i x | i > d       = [x]
               | otherwise = n ↑ i : go (i + 1) x
```

Here, d is the degree of the polynomial, n is the value to which the polynomial is applied and x is the result, *i.e.* the value from the sequence. The local function *go* repeats from 0 to d, raising n, the input value, to the current weight and adding it to the resulting list. At the end, when we have reached a value greater than the degree, we add the value from the sequence as known constant yielding one line of the system of linear equations.

When we apply *genCoeff* on the the sequence generated by x^4, we would have:

genCoeff 4 0 0 resulting in $[1, 0, 0, 0, 0, 0]$
genCoeff 4 1 1 resulting in $[1, 1, 1, 1, 1, 1]$
genCoeff 4 2 16 resulting in $[1, 2, 4, 8, 16, 16]$
genCoeff 4 3 81 resulting in $[1, 3, 9, 27, 81, 81]$
genCoeff 4 4 256 resulting in $[1, 4, 16, 64, 256, 256]$

Note that the results are very regular: we see constant 1 in the first column, the natural numbers in the first column, the squares in the third, the cubes in the fourth and the powers in the fifth and sixth column. This are just the values for x^i, for $i \in \{0 \dots 4\}$. Since the value in the sixth column, the one we took from the sequence, equals the value in the fifth column, we can already guess that the polynomial is simply x^4. Here is another sequence, generated by a secret polynomial:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204

We compute the difference lists using *dengine* as *ds* and compute the degree of the polynomial using *length ds*. The result is 4. Now we call *genCoeff* on the first four elements of the sequence:

genCoeff 4 0 14 resulting in $[1,0,0,0,0,14]$
genCoeff 4 1 62 resulting in $[1,1,1,1,1,62]$
genCoeff 4 2 396 resulting in $[1,2,4,8,16,396]$
genCoeff 4 3 1544 resulting in $[1,3,9,27,81,1544]$
genCoeff 4 4 4322 resulting in $[1,4,16,64,256,4322]$

We use *genCoeff* to create a matrix representing the entire system of equations:

```haskell
findCoeffs :: [[Zahl]] -> [Zahl] -> L.Matrix Zahl
findCoeffs ds seq = L.M (go 0 seq)
  where d = fromIntegral (length ds)
          go _ [] = []
          go n (x : xs) | n > d       = []
                        | otherwise = genCoeff d n x : go (n + 1) xs
```

The function *findCoeffs* receives the list of difference sequences created by *dengine* and the original sequence. It computes the degree of the generating polynomial as *length ds* and, then, it goes through the first d elements of the sequence calling *genCoeff* with d, the known input value, n, and x, the element of the sequence. For the sequence generated by x^4, we obtain $M\ [[1,0,0,0,0,0],[1,1,1,1,1,1],[1,2,4,8,16,16],[1,3,9,27,81,81],[1,4,16,64,256,256]]$, which corresponds to the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 & 16 \\ 1 & 3 & 9 & 27 & 81 & 81 \\ 1 & 4 & 16 & 64 & 256 & 256 \end{pmatrix}$$

For the sequence of the unknown polynomial, we obtain $M\ [[1,0,0,0,0,14],|\ [1,1,1,1,1,62],[1,2,4,8,16,396],[1,3,9,27,81,1544],[1,4,16,64,256,4322]]$, which corresponds to the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 14 \\ 1 & 1 & 1 & 1 & 1 & 62 \\ 1 & 2 & 4 & 8 & 16 & 396 \\ 1 & 3 & 9 & 27 & 81 & 1544 \\ 1 & 4 & 16 & 64 & 256 & 4322 \end{pmatrix}$$

The next steps are simple. We create the echelon form and solve by back-substitution. The following function puts all the bits together to find the generating polynomial:

$$\begin{array}{l} \mathit{findGen} :: [[\mathit{Zahl}]] \to [\mathit{Zahl}] \to [\mathit{Quoz}] \\ \mathit{findGen}\ \mathit{ds} = L.\mathit{backsub} \circ L.\mathit{echelon} \circ \mathit{findCoeffs}\ \mathit{ds} \end{array}$$

Applied on the difference list and the sequence generated by x^4, *findGen* yields: $[0\%1, 0\% 1, 0\,\%\,1, 0\,\%\,1, 1\,\%\,1]$, which indeed corresponds to the polynomial x^4. For the sequence generated by the unknown polynomial, we get: $[14\ \%\ 1, 9\ \%\ 1, 11\ \%\ 1, 16\ \%\ 1, 12\ \%\ 1]$, which corresponds to the polynomial $12x^4 + 16x^3 + 11x^2 + 9x + 14$. Let us test:

mapply $(P\ [14, 9, 11, 16, 12])\ [0\,..\,10]$ yields:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204,

which indeed is the same sequence we saw above!

Now, what about the differences generated by the difference engine? Those, too, are sequences of numbers. Are there polynomials that generate those sequences? The first difference sequence of our formerly unknown polynomial is

48,334,1148,2778,5512,9638,15444,23218,33248,45822

The next three difference sequences could be derived from this sequence – so, we can assume that this sequence is generated by a polynomial of degree 3. Let us see what *findGen* (*tail ds*) (*head ds*) yields with *ds* being the list of difference sequences of that polynomial: $[48\ \%\ 1, 118\ \%\ 1, 120\ \%\ 1, 48\ \%\ 1]$, which corresponds to the polynomial $48x^3 + 120x^2 + 118x + 48$. Let us test again:

mapply $(P\ [48, 118, 120, 48])\ [0\,..\,10]$ yields:

48,334,1148,2778,5512,9638,15444,23218,33248,45822,61228

The next difference sequence should then be generated by a polynomial of degree 2. We try with **let** $\mathit{ds}' = \mathit{tail}\ \mathit{ds}$ **in** *findGen* $(\mathit{tail}\ \mathit{ds}')\ (\mathit{head}\ \mathit{ds}')$ and get $[286\%1, 384\%1, 144\% 1]$, which corresponds to the polynomial $144x^2 + 384x + 286$.

mapply $(P\ [286, 384, 144])\ [0\,..\,10]$ yields:

286,814,1630,2734,4126,5806,7774,10030,12574,15406,18526

which, indeed, is the third difference sequence.

Finally, the last but one sequence, the last that is not constant, should be generated by a polynomial of degree 1. We try with **let** $\mathit{ds}'' = \mathit{tail}\ (\mathit{tail}\ \mathit{ds})$ **in** *findGen* $(\mathit{tail}\ \mathit{ds}'')\ (\mathit{head}\ \mathit{ds}'')$ and get $528\ \%\ 1, 288\ \%\ 1$ which corresponds to the polynomial $288x + 528$.

mapply $(P\ [528, 288])\ [0\,..\,10]$ yields:

528,816,1104,1392,1680,1968,2256,2544,2832,3120,3408 which, again is the expected difference sequence.

The differences are closely related to the tremendously important concept of the *derivative* of a function. The derivative of a polynomial π of degree n is a polynomial π' of degree $n-1$ that measures the *rate of change* or *slope* of π. The derivative expresses the rate of change precisely for any point in π. We will look at this with much more attention in the next chapter; the third part will then be entirely dedicated to derivatives and related concepts.

The difference sequences and the polynomials that generate them are also a measure of the rate of change. Actually, the difference between two points *is* the rate of change of that polynomial between those two points. The difference, however, is a sloppy measure.

Without going into too much detail here, we can quickly look at how the derivative of a polynomial is computed, which, in fact, is very easy. For a polynomial of the form

$$ax^n + bx^m + \dots + cx + d,$$

the derivative is

$$nax^{n-1} + mbx^{m-1} + \dots + c.$$

In other words, we drop the last term (which is the first term in our Haskell representation of polynomials) and, for all other terms, we multiply the term by the exponent and reduce the exponent by one.

The derivative of the polynomial x^4, for instance, is $4x^3$; in the notation of our polynomial type, we have $P\ [0,0,0,0,1]$ and its derivative $P\ [0,0,0,4]$. The derivative of $4x^3$ is $12x^2$, whose derivative then is $24x$, whose derivative is just 24. The deriviative of our polynomial

$$12x^4 + 16x^3 + 11x^2 + 9x + 14$$

is

$$48x^3 + 48x^2 + 22x + 9.$$

Note that the first term equals the first term of the polynomial that we identified as the generator of the first difference sequence. Indeed, the differences are sloppy as a measure for the rate of change – but they are not completely wrong!

Here is a function to compute the derivative:

```
derivative :: (Eq a, Num a, Enum a) ⇒ Poly a → Poly a
derivative (P as) = P (cleanz (go $ zip [1 ..] (drop 1 as)))
  where go []            = []
        go ((x, c) : cs) = (x ∗ c) : go cs
```

What is the sequence generated by the derivative of our polynomial? Well, we define the derivative as **let** $p' =$ *derivative* $(P\ [14, 9, 11, 16, 12])$, which is $P\ [9, 22, 48, 48]$, apply it using *mapply* $p'\ [0\,..\,10]$ and see:

9,127,629,1803,3937,7319,12237,18979,27833,39087,53029

Quite different from the first difference sequence we saw above!

What about the second derivative? we define as **let** $p'' =$ *derivative* p' and get $P\ [22, 96, 144]$. This polynomial creates the sequence

22,262,790,1606,2710,4102,5782,7750,10006,12550,15382

The next derivative, **let** $p''' =$ *derivative* p'', is $P\ [96, 288]$ and generates the sequence

96,384,672,960,1248,1536,1824,2112,2400,2688,2976.

You can already predict the next derivative, which is a polynomial of degree 0: it is $P\ [288]$. This is a constant polynomial and will generate a constant sequence, namely the sequence 288. That, however, was also the constant sequence generated by the difference engine. Of course, when the rate of change is the same everywhere in the original polynomial, then precision does not make any difference anymore. The two methods shall come to the same result.

Consider the simple polynomial x^2. It generates the sequence

$$0, 1, 4, 9, 16, 25, 36, 49, \dots$$

The differences are

$$1, 3, 5, 7, 9, 11, 13, \dots$$

The differences of this list are all 2.

The derivative of x^2 is $2x$. It would generate the sequence

$$0, 2, 4, 6, 8, 10, 12, 14, \dots$$

which does not equal the differences. However, we can already see that the derivative of $2x$, 2, is constant and generates the constant sequence

$$2, 2, 2, 2, 2, 2, 2, 2, \ldots$$

8.4. Differences and Binomial Coefficients

The ingenious Isaac Newton studied the relation between sequences and their differences intensely and came up with a formula. Before we go right to it, let us observe on our own. The following table shows, in the first line, the value of n, *i.e.* the value to which the polynomial is applied; in the second line, we see the result for this n; in the first column we have the first value from the sequence and from the difference lists:

	0	1	2	3	4
	14	62	396	1544	4322
14	1	1	1	1	1
48	0	1	2	3	4
286	0	0	1	3	6
528	0	0	0	1	4
288	0	0	0	0	1

What we see in the cells of the table are factors. With their help, we can compute the values in the sequence by formulas of the type:

$$\begin{array}{llllllllllllllllllll}
1 & \times & 14 & & & & & & & & & & & & & & & & = & 14\\
1 & \times & 14 & + & 1 & \times & 48 & & & & & & & & & & & & = & 62\\
1 & \times & 14 & + & 2 & \times & 48 & + & 1 & \times & 286 & & & & & & & & = & 396\\
1 & \times & 14 & + & 3 & \times & 48 & + & 3 & \times & 286 & + & 1 & \times & 528 & & & & = & 1544\\
1 & \times & 14 & + & 4 & \times & 48 & + & 6 & \times & 286 & + & 4 & \times & 528 & + & 1 & \times\ 288 & = & 4322
\end{array} \tag{8.13}$$

The next question would then be: what are those numbers? But, here, I have to ask you to look a bit more closely at the table. What we see is left-to-right:

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}$$

Those are binomial coefficients! Indeed. We could rewrite the table as

	0	1	2	3	4
	14	62	396	1544	4322

14	$\binom{0}{0}$	$\binom{1}{0}$	$\binom{2}{0}$	$\binom{3}{0}$	$\binom{4}{0}$
48	$\binom{0}{1}$	$\binom{1}{1}$	$\binom{2}{1}$	$\binom{3}{1}$	$\binom{4}{1}$
286	$\binom{0}{2}$	$\binom{1}{2}$	$\binom{2}{2}$	$\binom{3}{2}$	$\binom{4}{2}$
528	$\binom{0}{3}$	$\binom{1}{3}$	$\binom{2}{3}$	$\binom{3}{3}$	$\binom{4}{3}$
288	$\binom{0}{3}$	$\binom{1}{4}$	$\binom{2}{4}$	$\binom{3}{4}$	$\binom{4}{4}$

If this were universally true, we could devise a much better prediction function. The one we wrote in the previous section has the disadvantage that we can only predict the next number in the sequence. To predict a value way ahead we need to generate number by number before we are there. With Newton's trick, we could compute any number in the sequence in one step.

All we have to do is to get the *head*s of the sequences and to calculate the formula:

$$\sum_{k=0}^{d} h_k \binom{n}{k}$$

where d is the degree of the polynomial, n the position in the sequence, *i.e.* the number to which we apply the polynomial, and h_k the head of the sequence starting to count with the original sequence as $k = 0$. The sixth value ($n = 5$) of the sequence would then be

$$14 \times \binom{5}{0} + 48 \times \binom{5}{1} + 286 \times \binom{5}{2} + 528 \times \binom{5}{3} + 288 \times \binom{5}{4},$$

which is

$$14 + 48 \times 5 + 286 \times 10 + 528 \times 10 + 288 \times 5,$$

which, in its turn, is

$$14 + 240 + 2860 + 5280 + 1440 = 9834,$$

which is indeed the next value in the sequence.

Here is an implementation:

```
newton :: Zahl → [[Zahl]] → [Zahl] → Zahl
newton n ds seq = sum ts
  where hs = getHeads seq ds
        ts = [h ∗ (choose n k) | (h, k) ← zip hs [0 . . n]]
getHeads :: [Zahl] → [[Zahl]] → [Zahl]
getHeads seq ds = map head (seq : ds)
```

To perform some experiments, here, as a reminder, are the first 14 numbers of the sequence generated by our polynomial $P\ [14, 9, 11, 16, 12]$:

14,62,396,1544,4322,9834,19472,34916,58134,91382,137204,198432,278186,379874

We set s = *mapply* $P\ [14, 9, 11, 16, 12]\ [0\,..\,10]$ and d = *dengine* s. Now we perform some tests:

newton 0 d s gives 14.
newton 1 d s gives 62.
newton 5 d s gives 9834.
newton 11 d s gives 198432.
newton 13 d s gives 379874.

The approach, hence, seems to work. But there is more. The function *newton* gives us a closed form to compute any number in the sequence, given that we have the beginning of that sequence and its difference lists. A closed form, however, is a generating formula – it is the polynomial that generates the entire sequence. We just need a way to make the formula implicit in *newton* explicit.

We can do that using our polynomial data type. When we can express the binomial coefficients in terms of polynomials and apply them to the formula used above, we will get the polynomial out that generates this sequence. Here is a function that does it:

```
bin2poly :: Zahl → Zahl → Poly Quoz
bin2poly h 0   = P [h % 1]
bin2poly h 1   = P [0, h % 1]
bin2poly h k   = P [h % (B.fac k)] ‘mul‘ go (k % 1)
  where go 1 = P [0, 1]
        go i = P [−(i − 1), 1] ‘mul‘ (go (i − 1))
```

The function receives two integers: the first one is a factor (the head) by which we multiply the resulting binomial polynomial and the second one is k in $\binom{n}{k}$. Note that we do not need n, since n is the unknown, the base of our polynomial.

If $k = 0$, the binomial is 1, since for all binomials: $\binom{n}{0} = 1$. We, hence, return a constant polynomial consisting of the factor. This corresponds to $h_0 \times \binom{n}{0}$. The result is just h. Note that we convert the coefficients to rational numbers, since that is the type the function is supposed to yield.

If $k = 1$, the binomial is n, since for all binomials: $\binom{n}{1} = n$. Because n is the base of the polynomial, n itself is expressed by $P\ [0, 1]$. This is just $n + 0$ and, hence, n. Since we multiply with h, the result in this case is $h \times n = hn$, or, in the language of our Haskell polynomials $P\ [0, h]$.

Otherwise, we go into the recursive *go* function. The function receives one rational number, namely k (which, de facto, is an integer) The base case is $k = 1$. In that case we yield $P\ [0, 1]$, which is just n. Otherwise, we create the polynomial $P\ [-(i-1), 1]$, that is $n - (k - 1)$ and multiply with the result of *go* applied to $i - 1$. The function, hence, creates the numerator of the fraction formula of the binomial coefficient:

$$n(n-1)(n-2)\dots(n-k+1).$$

The result of the function is then multiplied by h divided by $k!$. The former, still, is some head from the difference sequences and the latter is the denominator of the fraction formula. We, thus, compute:

$$\frac{hn(n-1)(n-2)\dots(n-k+1)}{k!}.$$

Now, we can use this formula represented by a polynomial to compute the generating polynomial. The function that does so has exactly the same structure as the *newton* function. The difference is just that it expresses binomial coefficients as polynomials and that it does not receive a concrete number n for which we want to compute the corresponding value (because we want to compute the formula generating all the values):

```
newtonGen :: [[Zahl]] → [Zahl] → Poly [Quoz]
newtonGen ds seq = sump ts
  where hs = getHeads seq ds
        ts = [bin2poly h k | (h, k) ← zip hs [0 .. n]]
        n  = fromIntegral (length $ ds)
```

When we call *newtonGen ds s*, *ds* still being the difference lists and *s* the sequence in question, we see:

$P\ [14 \% 1, 9 \% 1, 11 \% 1, 16 \% 1, 12 \% 1]$,

which we immediately recognise as our polynomial $12x^4 + 16x^3 + 11x^2 + 9x + 14$.

For another test, we apply the monomial x^5 as

let *s* = *mapply* (*P* [0, 0, 0, 0, 0, 1]) [0 .. 10] **in** *newtonGen* (*dengine s*) *s*

and see

$P\ [0 \% 1, 0 \% 1, 0 \% 1, 0 \% 1, 0 \% 1, 1 \% 1]$, which is indeed the polynomial x^5.

But now comes the hard question: why does that work at all???

To answer this question, we should make sure to understand how Newton's formula works. The point is that we restrict ourselves to the heads of the sequences as basic building blocks. When we compute some value x_n in the sequence, we need to recursively compute x_{n-1} and the difference between x_{n-1} and x_n and add them together. Let us build a model that simulates this approach and that allows us to reason about what is going on more easily.

We use as a model a polynomial of degree 3; that model is sufficiently complex to simulate the problem completely and is, on the other hand, somewhat simpler than a model based on a polynomial of degree 4, like the one we have studied above.

The model consists of a data type:

```
data Newton = H | X | Y | Z
  deriving (Show, Eq)
```

The *Newton* type has four constructors: *H* represents the head of the original sequence; *X* is the head of the first difference list; *Y* is the head of the second difference list and *Z* is the constant element repeated in the last difference list. (Remember that a polynomial of degree 3 generates 3 difference lists.)

The model also contains a function to compute positions in the sequence. This function, called *cn* (for "computeNewton"), takes two arguments: a *Newton* constructor and an integer. The integer tells us the position we want to compute starting with the head $H = 0$:

```
cn :: Newton → Natural → [Newton]
cn H 0 = [H]
cn H n = cn H (n − 1) ++ cn X (n − 1)
```

When we want to compute the first element in the sequence, *cn H* 0, we just return [*H*]. When we want to compute any other number, we recursively call *cn H* $(n-1)$, which computes the previous data point, and add *cn X* $(n-1)$, which computes the difference between n and $n-1$. Here is how we compute the difference:

```
cn X 0 = [X]
cn X n = cn X (n − 1) ++ cn Y (n − 1)
```

If we need the first difference, *cn X* 0, we just return [*X*]. Otherwise, we call *cn X* $(n-1)$, this computes the previous difference, and compute *cn Y* $(n-1)$, the difference between the previous and the current difference. Here is how we compute the difference of the difference:

```
cn Y 0 = [Y]
cn Y n = Z : cn Y (n − 1)
```

If we need the first difference, *cn* Y 0, we just return $[Y]$. Otherwise, we compute the previous difference *cn* Y $(n-1)$ adding Z, the constant difference, to the result.

The simplest case is of course computing the first in the sequence. This is just:

cn H 0, which yields $[H]$.

Computing the second in the sequence is slightly more work:

cn H 1 goes to
cn H 0 ++ *cn* X 0 which is
$[H]$ ++ $[X]$.

We, hence, get $[H, X]$. That is the head of the sequence plus the head of the first difference list.

Computing the third in the sequence

cn H 2 calls
cn H 1 ++ *cn* X 1, which is
cn H 0 ++ *cn* X 0 and *cn* X 0 ++ *cn* Y 0.

We hence get $[H, X, X, Y]$. This is the head of the original sequence plus the head of the first difference sequence (we are now at H 1) plus this difference plus the first of the second difference sequence.

This looks simple, but already after a few steps, the result looks weird. For *cn* H 5, for example, we see

$[H, X, X, Y, X, Y, Z, Y, X, Y, Z, Y, Z, Z, Y, X, Y, Z, Y, Z, Z, Y, Z, Z, Z, Y]$,

which is somewhat confusing. The result, however, is correct. When we generate a random polynomial of degree 3, say, P $[2, 28, 15, 22]$, this is the polynomial $22x^3 + 15x^2 + 28x + 2$, we get the sequence 2, 67, 294, 815, 1762, 3267, 5462, 8479, 12450, 17507, 23782. We now define a function that substitutes the symbols of our model by the heads of the sequence and the difference lists:

```
new2a :: (a, a, a, a) → Newton → a
new2a (h, x, y, z) n = case n of
  H → h
  X → x
  Y → y
  Z → z
subst :: (a, a, a, a) → [Newton] → [a]
subst as = map (new2a as)
```

The head of the sequence is 2; the head of the difference sequences are 65, 162 and 132. We call the function as *subst* $(2, 65, 162, 132)$ (*cn* H 5) and see

```
2,65,65,162,65,162,132,162,65,162,132,162,132,132,162,
65,162,132,162,132,132,162,132,132,132,162.
```

When we sum this together, *sum* (*subst* $(2, 65, 162, 132)$ (*cn* H 5)), we get 3267, which is indeed the number appearing at position 5 in the sequence (starting to count with 0).

We implement one more function: *ccn*, for “count cn”:

```
ccn :: [Newton] → (Int, Int, Int, Int)
ccn ls = (length (filter (≡ H) ls),
          length (filter (≡ X) ls),
          length (filter (≡ Y) ls),
          length (filter (≡ Z) ls))
```

When we apply this function, *e.g.* *ccn* (*cn* H 3), we see:

```
(1, 3, 3, 1)
```

The binomial coefficients $\binom{3}{k}$, for $k \in \{0 \dots 3\}$.

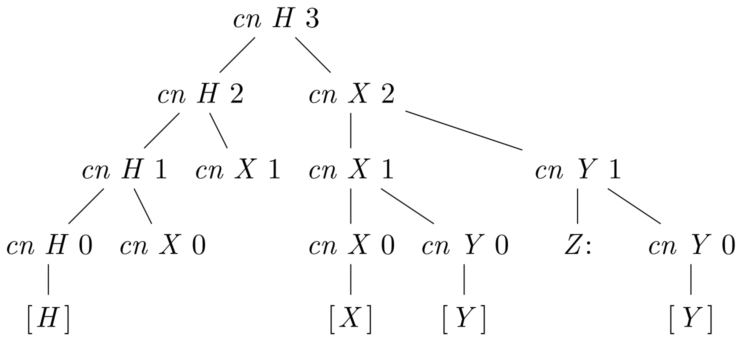
To see some more examples we call *map* (*ccn* ∘ *cn* H) $[4 \,.\,. 10]$ and get

```
[(1, 4, 6, 4),
(1, 5, 10, 10),
(1, 6, 15, 20),
(1, 7, 21, 35),
(1, 8, 28, 56),
(1, 9, 36, 84),
(1, 10, 45, 120)]
```

What we see, in terms of the table we used above, is

	0	1	2	3
	n_0	n_1	n_2	n_3
H	$\binom{0}{0}$	$\binom{1}{0}$	$\binom{2}{0}$	$\binom{3}{0}$
X	$\binom{0}{1}$	$\binom{1}{1}$	$\binom{2}{1}$	$\binom{3}{1}$
Y	$\binom{0}{2}$	$\binom{1}{2}$	$\binom{2}{2}$	$\binom{3}{2}$
Z	$\binom{0}{3}$	$\binom{1}{3}$	$\binom{2}{3}$	$\binom{3}{3}$

So, why do we see binomial coefficients and can we prove that we will always see binomial coefficients? To answer the first question, we will analyse the execution tree of *cn*. Here is the tree for *cn* H 3:



On the left-hand side of the tree, you see the main execution path calling $cn\ H\ (n-1)$ and $cn\ X\ (n-1)$ on each level. The sketch expands $cn\ X$ only for one case, namely the top-level call $cn\ X\ 2$ on the right-hand side. Otherwise, the tree would be quite confusing.

Anyway, what we can see:

- Any top-level call of type $cn\ A$ (for $A \in \{H, X, Y\}$) creates only one A; we therefore have always exactly one H.
- Every call to $cn\ H\ n$, for $n > 0$, calls one instance of $cn\ X$. We therefore have exactly n X.
- Every call to $cn\ X\ n$, for $n > 0$, calls one instance of $cn\ Y$. We therefore have exactly n Y per $cn\ X\ n$, $n > 0$.
- Every call to $cn\ Y\ n$, for $n > 0$, creates one Z.
- The call to $cn\ X\ 1$ would expand to $cn\ X\ 0 +\!\!+\ cn\ Y\ 0$; it would, hence, create one more X and one more Y.
- The call to $cn\ X\ 0$ would create one more X.
- This execution, thus, creates 1 H, 3 X, 3 Y and 1 Z.

We now prove by induction that if a call to $cn\ H\ n$ creates

$$\binom{n}{0}H, \binom{n}{1}X, \binom{n}{2}Y \text{and} \binom{n}{3}Z$$

(and the previous calls to $cn\ H\ (n-1)$, $cn\ H\ (n-2)$, …, $cn\ H\ 0$ created similar patterns including the binomial coefficients), then $cn\ H\ (n+1)$ creates

$$\binom{n+1}{0}H, \binom{n+1}{1}X, \binom{n+1}{2}Y \text{and} \binom{n+1}{3}Z.$$

Note that the number of H does not increase, because, as observed, each top-level call to *cn A n* creates exactly one A. If *cn H n* creates one H, *cn H* $(n+1)$ creates exactly one H, too. We conclude that we create $\binom{n+1}{0}H$ as requested.

When we call *cn H* $(n+1)$, we will call *cn H n*. We, therefore, create all instances of X created by *cn H n* plus those created in the first level of *cn H* $(n+1)$. This new level calls *cn X n* exaclty once, which creates one X (because any top-level call to *cn A n* creates exactly one A). We, hence, create one X more. This, however, is $\binom{n}{0} + \binom{n}{1} = \binom{n+1}{1}$ according to Pascal's Rule. We conclude that we create $\binom{n+1}{1}X$ as requested.

Since we call *cn H n*, when we call *cn H* $(n+1)$, we also create all instances of Y that were created by *cn H n*. We additionally create all instances of Y that are created by the new call to *cn X n*. This, in its turn, calls n instances of *cn Y*. Since $n = \binom{n}{1}$ and any top-level call to *cn Y n* creates exactly one Y, we create $\binom{n}{1} + \binom{n}{2} = \binom{n+1}{2}Y$ as requested.

Finally, since we call *cn H n*, when we call *cn H* $(n+1)$, we also create all instances of Z that were created before. But we call one more instance of *cn X n*, which creates a certain amount of new Z. How many? We create again all Z that were created anew by *cn H n*, those that did not exist in *cn H* $(n-1)$. Let us call the number of Z created by *cn H n* z_n and the number of Z created by *cn H* $(n-1)$ z_{n-1}. The number of Z created anew in *cn H n* is then $z_n - z_{n-1}$.

But since, in *cn H* $(n+1)$, we call *cn X* one level up, more Z are created than before. All calls to *cn Y* 0, those that did not create a new Z in *cn H n*, are now called as *cn Y* 1 and, hence, create a Z that was not created before. The calls to *cn Y* 0 create Y that were not created by *cn H* $(n-1)$. We, therefore, need to add to the number of Z the number of Y that did not exist in *cn H* $(n-1)$. We use the same convention as for Z, *i.e.* the number of Y created anew in *cn H n* is $y_n - y_{n-1}$. The number of additional Z created by the additional call to *cn X n*, hence, is

$$y_n - y_{n-1} + z_n - z_{n-1}$$

But we are dealing with binomial coefficients. We, therefore, have $z_n = y_{n-1} + z_{n-1}$ by Pascals' Rule applied backwards. When we substitute this back, we get

$$y_n - y_{n-1} + y_{n-1} + z_{n-1} - z_{n-1},$$

which simplifies to y_n, *i.e.* the number of instances of Y created by *cn H n*. In other words: the number of Z we additionally create in *cn H* $(n+1)$ is the number of Y in *cn H n*. So, the complete number of Z we have in *cn H* $(n+1)$ is the number of Y in *cn H n* plus the number Z in *cn H n*. Since the number of Y is $\binom{n}{2}$ and the number of Z is $\binom{n}{3}$, we now have $\binom{n}{2} + \binom{n}{3} = \binom{n+1}{3}$ according to Pascal's Rule as requested and this completes the proof. □

8.5. Umbral Calculus

8.6. Roots

In the previous sections, we looked at the results, when applying polynomials to given values. That is, we applied a polynomial $\pi(x)$ to a given value (or sequence of values) for x and studied the result $y = \pi(x)$. Now we are turning this around. We will look at a given y and ask which value x would create that y. In other words, we look at polynomials as equations of the form:

$$a_nx^n + a_{n-1}x^{n-1} + \dots + a_0 = a \tag{8.14}$$

and search for ways to solve such equations. In the focus of this investigation is usually the special case $a = 0$, *i.e.*

$$a_nx^n + a_{n-1}x^{n-1} + \dots + a_0 = 0. \tag{8.15}$$

The values for x fulfilling this equation are called the *root*s of the polynomial. A trivial example is x^2, whose root is 0. A slightly less trivial example is $x^2 - 4$, whose roots are $x_1 = -2$ and $x_2 = 2$, since

$$(-2)^2 - 4 = 4 - 4 = 0$$

and

$$2^2 - 4 = 4 - 4 = 0.$$

Note that these examples are polynomials of even degree. Polynomials of even degree do not need to have any roots. Since even powers are always positive (or zero), negative values are turned into positive numbers and, since the term of highest degree is even, the whole expression may always be positive. This is true for the polynomial $x^2 + 1$. Since all negative values are transformed into positive values by x^2, the smallest value that we can reach is the result for $x = 0$, which is $0 + 1 = 1$.

On the other hand, even polynomials may have negative values, namely when they have terms with negative coefficients that, for smaller numbers, result in numbers that are greater than those resulting from the term of highest degree. The polynomial $x^2 - 4$, for instance, is negative in the interval $]-2\dots 2[$. It, therefore, must have two roots: one at -2, where the polynomial results become negative, and the other at 2, where the polynomial results become positive again.

Odd polynomials, by contrast, usually have negative values, because the term with the highest degree may result in a negative or a positive number depending on the signedness of the input value and that of the coefficient. The trivial polynomial x^3, for instance, is negative for negative values and positive for positive values. The slightly less trivial polynomial $x^3 + 9$ has its root at -3, while $x^3 - 9$ has its root at 3.

In summary, we can say that even polynomials do not necessarily have negative values and, hence, do not need to have a root. Odd polynomials, on the other hand, usually have both, negative and positive values, and, hence, must have a root.

Those are strong claims. They are true, because polynomials belong to a specific set of *functions*, namely *continuous* functions. That, basically, means that they have no *holes*, *i.e.* for any value x of a certain number type there is a result y of that number type. For instance, when the coefficients of the polynomial are all integers and the x-value is an integer, then the result is an integer, too. When the polynomial is defined over a field (all coefficients are part of that field and the values to which we apply the polynomial lie in that field), then the result is in that field, too. Rational polynomials, for instance, have rational results. Real polynomials have real results.

Furthermore, the function does not "jump", *i.e.* the results grow with the input values – not necessarily at the same rate, in fact, for polynomials of degree greater than 1, the result grows much faster than the input – but the growth is regular.

These properties appear to be "natural" at the first sight. But there are functions that do not fulful these criteria. In the next chapter, when we properly define the term *function*, we will actually see functions with holes and jumps.

The reason that polynomials behave regularily is that we only use basic arithmetic functions in their definition: we add, multiply and raise to powers. Those operations together with the integers form monoids and, with rational and real numbers, they form groups. Both, monoids and groups, are closed over their base set. We can therefore be sure that, for any input value from the base set, the result is in the same base set, too.

Furthermore, the form of polynomials guarantees that they develop in a certain way. For very large numbers (negative or positive), it is the term with the greatest exponents, *i.e.* the degree of the polynomial, that most significantly determines the outcome, that is, the result for very large numbers approaches the result for the term with the largest exponent. For smaller values, however, the terms of lower degree have stronger impact. The terms "large" and "small", here, must be understood relative to the coefficients. If the coefficients are very large, the values to which the polynomial is applied must be even larger to approach the result for the first term.

There are also polynomials with a quite confusing behaviour that make it hard to guess the roots, for instance, Wilkinson's polynomial named for James Hardy Wilkinson (1919 – 1986), an American mathematician and computer scientist. The Wilkinson polynomial is defined as

$$w(x) = \prod_{i=1}^{20} (x - i). \tag{8.16}$$

We can generate it in terms of our polynomial type as

```
wilkinson :: (Num a, Enum a, Show a, Eq a) ⇒ Poly a
wilkinson = prodp mul [P [−i, 1] | i ← [1 . . 20]]
```

It looks like this:

```
P [
2432902008176640000, −8752948036761600000, 13803759753640704000,
−12870931245150988800, 8037811822645051776, −3599979517947607200,
1206647803780373360, −311333643161390640, 63030812099294896,
−10142299865511450, 1307535010540395, −135585182899530, 11310276995381,
−756111184500, 40171771630, −1672280820, 53327946, −1256850, 20615, −210, 1]
```

The first terms are

$$x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} \dots$$

When we apply Wilkinson's polynomial to the integers $1 \dots 10$, we see:

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2432902008176640000,
51090942171709440000, 562000363888803840000, 4308669456480829440000,
25852016738884976640000,
```

which looks very confusing. When we try non-integers, we see

apply wilkinson 0.9 is 1.7213 . . .
apply wilkinson 1.1 is −8.4600 . . .
apply wilkinson 1.9 is −8.1111 . . .
apply wilkinson 2.1 is 4.9238 . . .

As we see, the results switch sign at the integers or, more precisely, at the integers in the interval $[1 \dots 20]$, which are the roots of Wilkinson's polynomial. Looking at the factors of the polynomial

$$(x - 1)(x - 2) \dots (x - 20),$$

this result is much less surprising, since, obviously, when any of these factors becomes 0, then the whole expression becomes 0. So, for the value $x = 3$, we would have

$$2 \times 1 \times 0 \times \cdots \times -17 = 0.$$

The results when applying the polynomial, however, look quite irregular and, on the first sight, completely unrelated to the coefficients. When we say that polynomials show a regular behaviour, that must be taken with a grain of salt. Anyway, that they behave like this gives rise to a number of simple methods to find roots based on approximation, at least when we start with a fair guess, which requires some knowledge about the rough shape of the polynomial in the first place.

These methods can be split into two major groups: *bracketing* methods and *open* methods. Bracketing methods start with two distinct values somewhere on the “left” and the “right” of the root. Bracketing methods, hence, require a pre-knowledge about where, more or less, a root is located. We then choose two values that limit this interval on the lower and on the upper side.

The simplest variant of bracketing is the *bisect* algorithm. It is very similar to Heron’s method to find the square root of a given number. We start with two values a and b and, on each step, we compute the average $(a + b)/2$ and substitute either a or b by this value depending on the side the value is located relative to the root. Here is an implementation:

```
bisect :: (Num a, Eq a, Ord a, Fractional a, Show a)
       ⇒ Poly a → a → a → a → a
bisect p t a b  | abs fc < abs t          = c
                | signum fc ≡ signum fa   = bisect p t c b
                | otherwise               = bisect p t a c
  where fa = apply p a
        fb = apply p b
        fc = apply p c
        c  = (a + b) / 2
```

The function receives four arguments. The first is the polynomial. The second is a tolerance. When we reach a result that is smaller than the tolerance, we return the result. a and b are the starting values.

We distinguish three cases:

- The result for the new value, c, is below the tolerance threshold. In this case, c is sufficiently close to the root and we yield this value.
- the sign of the result for the new value equals the sign of a. Then we replace a by c.

- the sign of the result for the new value equals the sign of b. In this case, we replace b by c.

We try *bisect* on the polynomial x^2 with the initial guess $a = -1$ and $b = 1$ (because we assume that the root should be close to 0) and a tolerance of 0.1:

$bisect\ (P\ [0,0,1])\ 0.1\ (-1)\ 1$

and see the correct result 0.0.

For the polynomial $x^2 - 4$, which has two roots, we try

$bisect\ (P\ [-4,0,1])\ 0.1\ (-3)\ (-1)$,

which yields -2 and

$bisect\ (P\ [-4,0,1])\ 0.1\ 1\ 3$,

which yields 2.

With Wilkinson's polynomial, however, we get a surprise:

$bisect\ wilkinson\ 0.1\ 0.5\ 1.5$,

for which we expect to find the root 1. But the function does not return. Indeed, when we try *apply wilkinson* 1.0, we see

1148.0,

a somewhat surprising result. Wilkinson used this polynomial to demonstrate the sensivity of coefficients to small differences in the input values. Using Haskell real numbers, The computation leads to a loss of precision in representing the terms. Indeed, considering terms raised to the 20^{th} power and multiplied by large coefficients, the number 1148 appears to be a tiny inprecision.

We can work around this, using rational numbers:

$apply\ wilkinson\ (1\ \%\ 1)$

gives without any surprise $0\ \%\ 1$. So, we try

$bisect\ wilkinson\ (1\ \%\ 10)\ (1\ \%\ 2)\ (3\ \%\ 2)$

and get the correct result $1\ \%\ 1$. The function with this parameters returns almost instantanious. That is because the average of 0.5 and 1.5 is already 1. The function finds the root in the first step. A more serious challenge is

$bisect\ wilkinson\ (1\ \%\ 10)\ (1\ \%\ 3)\ (3\ \%\ 2)$,

which needs more than one recursion. The function, now, runs for a short while and comes up with the result

1729382256910270463 % 1729382256910270464,

which is pretty close to 1 and, hence, the correct result.

The most widely known open method is Newton's method, also called Newton-Raphson method. It was first developed by Newton in about 1670 and, then, by Joseph Raphson in 1690. Newton's version was probably not known to Raphson, since Newton did not publish his work. Raphson's version, on the other hand, is simpler and, therefore, usually preferred.

Anyway, the method starts with only one approximation and is therefore not a bracketing method. The approximation is then applied to the polynomial π and the derivative of that polynomial, π'. Then, the quotient of the results, $\frac{\pi(x)}{\pi'(x)}$ is computed and subtracted from the initial guess. Here is an implementation:

```
newguess :: (Num a, Eq a, Ord a, Enum a, Fractional a)
            ⇒ Poly a → Natural → a → a → a
newguess p m t a | abs pa < t = a
                 | m ⩽ 0      = a
                 | otherwise  = newguess p (m − 1) t (a − pa / p′a)
  where p′  = derivative (∗) p
        pa  = apply p a
        p′a = apply p′ a
```

The function receives four parameters. The polynomial p, the natural number m, the tolerance t and the initial guess a. The natural number m is a delimiter. It is not guaranteed that the value increases in precision with always more repetitions. It may get worse at some point. It is therefore useful to restrict the number of iterations.

The function terminates when we have reached either the intended precision or the number of repetitions, m. Otherwise, we repeat with $m - 1$ and with $a - \frac{\pi(a)}{\pi'(a)}$.

For the polynomial $x^2 - 4$, we call first

newguess (*P* [−4, 0, 1]) 10 0.1 1

and get 2.00069..., which is very close to the known root 2. For the other root we call

newguess (*P* [−4, 0, 1]) 10 0.1 (−1)

and get the equally close result −2.00069... For the Wilkinson polynomial, we call

newguess wilkinson 10 (0.0001) 1.5

and get 1.99999..., which is very close to the real root 2. We can further improve precision by increasing the number of iterations:

newguess wilkinson 20 (0.0001) 1.5

The difference is at the 12^{th} decimal digit.

Note that the Newton-Raphson method is not only more precise (that is: converges earlier with a good result), but also more robust against real representation imprecision.

To understand why this method works at all, we need to better understand what the derivative is. We will come back to this issue in the next chapter. In the strict sense, the derivative does not belong here anyway, since the concept of derivative is analysis, not algebra. Both kinds of methods, the bracketing and the open methods, in fact, come from numerical analysis. They do not have the "look and feel" of algebraic methods. So, how would an algebraist tackle the problem of finding the roots of a polynomial?

One possibility is factoring. Polynomials may be represented as the product of their factors (just like integers). We have experienced with Wilkinson's polynomial that the factor representation may be much more convenient that the usual representation with coefficients. Wilkinson's polynomial expressed as a product was just

$$w(x) = \prod_{i=1}^{20} (x - i), \tag{8.17}$$

i.e.: $(x-1)(x-2)\dots(x-20)$.

As for all products, when one of the factors is zero, then the whole product becomes zero. For the root problem, this means that, when we have the factors, we can find a value for x, so that any of the factors becomes zero and this value is then a root, because it makes the product and, as such, the whole polynomial zero. Any integer in the range $[1\dots20]$ would make one of the factors of Wilkinson's polynomial zero. The integers $[1\dots20]$ are therefore the roots of Wilkinson's polynomial.

Factoring polynomials, however, is an advanced problem in its own right and we will dedicate the next section to its study. Anyway, what algebraists did for centuries was to find formulas that would yield the roots for any kind of polynomials. In some cases they succeeded, in particular for polynomials of degrees less than 5. For higher degrees, there are not such formulas. This dicovery is perhaps much more important than the single formulas developed over the centuries for polynomials of the first four degrees. In fact, the concepts that led to the discovery are the foundations of modern (and postmodern) algebra.

But first things first. To understand why there cannot be general formulas for solving polynomials higher degrees, we need to understand polynomials much better. First, we will look at the formula to solve polynomials of the second degree. Note that we skip the first degree, since finding the roots of polynomials of the first degree is just solving linear equations.

8.7. Vieta's Formula

8.8. Factoring Polynomials

Polynomials can be factored in different contexts, for instance a field or the integers (which, as you may remember, do not form a field, but a ring). These contexts can be generalised to what is called a *unique factorisation domain.* A unique factorisation domain is a commutative ring R, where

- $uv \neq 0$, whenever $u, v \in R$ and $u \neq 0$ and $v \neq 0$;
- every nonzero element is a *unit*, a *prime* or can be uniquely represented as a product of primes;
- every unit u has an inverse v, such that $uv = 1$.
- a prime p is a nonunit element for which an equation of the form $p = qr$ is true, only if either q or r is a unit.

The integers form a unique factorisation domain, with the units 1 and -1 and the primes $\pm 2, \pm 3, \pm 5, \pm 7, \dots$ We can easily verify that 1 and -1 obey to the definition of unit, when we assume that each one is its own inverse. We can also agree that the primes are primes in the sense of the above definition: for any prime in $p \in \mathbb{Z}$, if $p = qr$, then either q or r must be a unit and the other must equal p. That is the definition of primes.

A field is trivially a unique factorisation domain without primes where all elements are units.

The simplest notion of factoring in such a domain is the factoring into *primitive part* and *content.* This, basically, splits a polynomial into a number (in the domain we are dealing with) and a *primitive polynomial.*

With the integers, the content is the GCD of the coefficients. For instance, the GCD of the coefficients of the polynomial $9x^5 + 27x^2 + 81$ is 9. When we divide the polynomial by 9 we get $x^5 + 3x^2 + 9$.

For rational numbers, we would choose a fraction that turns all coefficients into integers that do not share divisors. The polynomial

$$\frac{1}{3}x^5 + \frac{7}{2}x^2 + 2x + 1,$$

for instance, can be factored dividing all coefficients by $\frac{1}{6}$:

$$\begin{array}{cccrr} \frac{1}{3} & \times & 6 & = & 2 \\ \frac{7}{2} & \times & 6 & = & 21 \\ 2 & \times & 6 & = & 12 \\ 1 & \times & 6 & = & 6 \end{array}$$

We, hence, get the product $\frac{1}{6}(2x^5 + 21x^2 + 12x + 6)$.

This, however, is not the end of the story. Consider the polynomial

$$3x^2 - 27.$$

We can factor this one into $3(x^2 - 9)$, with the second part being primitve: the GCD of its coefficients is 1. But we can factor it further. Obviously, we have

$$x^2 - 9 = (x - 3)(x + 3). \tag{8.18}$$

The complete factorisation of the polynomial $3x^2 - 27$, hence, is $3(x - 3)(x + 3)$.

For factoring primitive polynomials manually, there are many different methods (most of which have a video on youtube). They share one property: they are highly inefficient, when it comes to polynomials of larger degrees or with big coefficients. They, basically, all use integer factorisation of which we know that it is extremely expensive in terms of computation complexity. Instead of going through all of them, we will here present a typical classical method, namely Kronecker's method.

Kronecker's method is a distinct-degree approach. That is, it searches for the factors of a given degree. We start by applying the polynomial to n distinct values, for n the degree of the polynomial plus 1. That is because, to represent a polynomial of degree d, we need $d + 1$ coefficients, *e.g.* $P\ [0, 0, 1]$ has three coefficients and represents the polynomial x^2, which is of degree two.

The rationale of applying the polynomial is the following: When the polynomial we want to factor generates a given set of values, then the product of the factors of that polynomial must generate the same values. Any factor must, hence, consist of divisors of those values. The number of integer divisors of those values, however, is limited. We can therefore afford, at least for small polynomials with small coefficients, trying all the combinations of the divisors.

We have already defined a function to find the divisors of a given number, when we discussed Euler's totient function. However, that function dealt with natural numbers only. We now need a variant that is able to compute negative divisors. It would be also nice if that function could give us not only the divisors, but additionally the additive inverse, *i.e.* the negation of the divisors, because, in many cases, we need to look at the negative alternatives too. Here is an implementation:

```
divs :: Zahl → [Zahl]
divs i | i < 0       = divs (−i)
       | otherwise = ds ++ map negate ds
  where ds = [d | d ← [1 .. i], rem i d ≡ 0]
```

The divisors are now combined to yield n-tuples with n still the degree of the factor plus one and each divisor representing one coefficient of the resulting polynomial. But before we can convert the n-tuples into polynomials, we need to create all possible permutations, since the polynomial $P\ [a, b]$ is not the same as $P\ [b, a]$ if $a \neq b$. From this we obtain a (potentially very large) list of n-tuples that we then convert into polynomials. From that list, we finally filter those polynomials for which p ‘*divp*‘ $k \equiv 0$, where p is the input polynomial and k the candidate in the list of polynomials. Here is an implementation (using lists instead of n-tuples):

```
kronecker :: Poly Zahl → [Zahl] → [Poly Quoz]
kronecker (P cs) is = nub [a | a ← as, snd (r ‘divp‘ a) ≡ P [0]]
  where ds = map divs is
        ps = concatMap perms (listcombine ds)
        as = map (P ∘ map fromInteger) ps
        r  = P [c % 1 | c ← cs]
```

The function takes two arguments. The first is the polynomial we want to factor and the second is the list of results obtained by applying the polynomial. We then get the divisors *ds*, create all possible combinations of the divisors and all possible permutations of the resulting lists. We then convert the coefficients to rational numbers (since we later use *divp*). Finally, we filter all polynomials that leave no remainder when the input polynomial is divided by any one of them.

There are two combinatorial functions, *perms* and *listcombine*. We have already defined *perms*, when discussing permutations. The function generates all permutations of a given list. The other function, *listcombine*, however, is new. It creates all possible combinations of a list of lists. Here is a possible implementation:

```
listcombine :: [[a]] → [[a]]
listcombine []        = []
listcombine ([]: _)   = []
listcombine (x : xs) = inshead (head x) (listcombine xs) ++
                       listcombine ((tail x) : xs)

inshead :: a → [[a]] → [[a]]
inshead x [] = [[x]]
inshead x zs = map (x:) zs
```

Let us try *kronecker* on some polynomials. First, we need to apply the input polynomial to get n results. For instance, we know that the polynomial $x^2 - 9$ has factors of first degree. We, therefore, apply it on two values: **let** *vs* = *mapply* (P $[-9, 0, 1]$) $[0, 1]$ and get for *vs*: $[-9, -8]$. Now we call *kronecker* (P $[-9, 0, 1]$) $[-9, -8]$ and get:

P $[3 \% 1, 1 \% 1]$
P $[3 \% 1, (-1) \% 1]$
P $[(-3) \% 1, 1 \% 1]$
P $[(-3) \% 1, (-1) \% 1]$

Those are the polynomials $x + 3$, $-x + 3$, $x - 3$ and $-x - 3$. By convention, we exclude the polynomials starting with a negative coefficients by factoring out -1 first. However, we can easily see that all of them are actually factors of $x^2 - 9$, since

$$(x + 3)(x - 3) = (x^2 - 9) \tag{8.19}$$

and

$$(-x + 3)(-x - 3) = (x^2 - 9). \tag{8.20}$$

Here is another example: $x^5 + x^4 + x^2 + x + 2$. We want to find a factor of degree 2, so we apply the polynomial to three values, say, $[-1, 0, 1]$. The result is $[2, 2, 6]$. We run *kronecker* (P $[2, 1, 1, 0, 1, 1]$) $[2, 2, 6]$ and, after a short while, we get:

P $[1 \% 1, 1 \% 1, 1 \% 1]$
P $[2 \% 1, 2 \% 1, 2 \% 1]$
P $[(-1) \% 1, (-1) \% 1, (-1) \% 1]$
P $[(-2) \% 1, (-2) \% 1, (-2) \% 1]$,

which corresponds to the polynomials x^2+x+1, $2x^2+2x+2$, $-x^2-x-1$ and $-2x^2-2x-2$. Ony the first one is a primitive polynomial. We can factor out 2 from the second one, leaving just the first one; polynomials three and four, simply, are the negative counterparts of one and two, so we can factor out -1 and -2, respectively, to obtain again the first one.

To check if the first one is really a factor of the input polynomial we divide:

P $[2, 1, 1, 0, 1, 1]$ ‘*divp*‘ P $[1, 1, 1]$
and get P $[2, -1, 0, 1]$, which corresponds to $x^3 - x + 2$. Indeed:

$$(x^2 + x + 1)(x^3 - x + 2) = x^5 + x^4 + x^2 + x + 2. \tag{8.21}$$

Kronecker's method is just a brute force search. It is obvious that it is not efficient and will fail with growing degrees and coefficients. Modern methods to factor polynomials use much more sophisticated techniques.

They are, in particular, based on modular arithmetic and make use of theorems that we have already discussed in the ring of integers. Polynomials with coefficients in a ring (or field) form a ring too, a polynomial ring. Theorems that hold in any ring, hence, hold also in a polynomial ring. We, therefore, do not need to prove them here again.

We will discuss the methods for factorising polynomials in a finite field in the next section. Let us here assume that we already knew such a method. We could then call it to factor a given polynomial in a finite field and then reinterpret the result for the domain we started with.

8.9. Factoring Polynomials in a finite Field

Famous factorisation algorithms using modular arithmetic are *Berlekamp's algorithm* developed by the American mathematician Elwyn Berlekamp in the late Sixties and the *Cantor-Zassenhaus algorithm* developed in the late Seventies and early Eighties by David Cantor, an American methematician, not to be confused with Georg Cantor, and Hans Zassenhaus (1912 – 1991), a German-American mathematician. We will here focus on Cantor-Zassenhaus, which is by today probably the most-used algorithm implemented in many computer algebra systems.

The contribution of Cantor-Zassenhaus, strictly speaking, is just one of several pieces. The whole approach is based on Euler's theorem, which, as you may remember, states that

$$a^{\varphi(n)} \equiv 1 \pmod{n}, \tag{8.22}$$

where $\varphi(n)$ is the totient function of n counting the numbers $1 \dots n-1$ that are coprime to n, *i.e.* that share no divisors with n.

Euler's theorem is defined as theorem over the ring of integers, which, by modular arithmetic, transforms into the finite field of the integers $0 \dots n-1$. Polynomial rings can be seen as extensions of the underlying ring (of integers). When we introduce modular arithmetic, that is, when we build polynomials on a finite field, they still constitute a ring, but now a ring built on top of a finite field. Notationally, this is usually expressed as $K[x]$, where K is a field and $K[x]$ the polynomial ring defined on top of K.

When we now take polynomials modulo a polynomial, we again get a finite field, this time a polynomial field of the form $K[x]/m$ (pronounced "over" m), where m is a polynomial. The point in doing this is that many properties of the original field K are preserved in $K[x]/m$ and Euler's theorem is one of them.

However, we need to redefine Euler's theorem to make clear what is meant by it in the new context. We are now dealing with the polynomial ring $K[x]$ and a polynomial $m \in K[x]$. Based on this, we can define the totient function as

$$\varphi(m) = |\{f \in K[x] : 0 \leq f \leq m \wedge \gcd(m, f) = 1\}|,$$

i.e. the cardinality of the set of all polynomials f less or equal than m that do not share divisors with m. For any such ring $K[x]$ and any $f \in K[x] : \gcd(m, f) = 1$, the following holds:

$$f^{\varphi(m)} \equiv 1 \pmod{m}. \tag{8.23}$$

This is easy to prove. The resulting structure $K[x]/(m)$ has a multiplicative group K_m^* (just as the integers $\pmod{n}$). The members of this group are all polynomials that do not share divisors with m and $\varphi(m)$ is the cardinality of this group. From $gcd(m, f) = 1$, it follows that $f_m = f \mod m \in K_m^*$. We, for sure, have $f_m^{\varphi(m)} \equiv 1 \pmod{m}$, since $\varphi(m)$ is the size of the group. This equivalence may hold also for other numbers, a, such that $f^a \equiv 1 \pmod{m}$, but according to Lagrange's theorem (that the cardinality of subgroups of G divides the cardinality of G), all these numbers a must divide $\varphi(m)$, the size of the group, and we unmistakenly have $f^{\varphi(m)} \equiv 1 \pmod{m} \quad \square$.

From this theorem, Fermat's little theorem follows. Let K be a field with q elements; when using arithmetic modulo a prime p, then K_m^* is the group of numbers $1 \dots p-1$, which has $q = p - 1$ elements. Note that, when we refer to the multiplicative group of this field, we usually refer only to the numbers $1 \dots p - 1$, *i.e.* $p - 1$ numbers. Now, let g be an *irreducible* polynomial, *i.e.* a non-constant polynomial that cannot be further factored and, hence, a "prime" in our polynomial ring, with degree d, $d > 0$. Then it holds for any polynomial f from this field

$$f^{q^d} \equiv f \pmod{g}. \tag{8.24}$$

We can prove this easily: We know that K has q elements. From this q elements we can create a limited number of polynomials. When you look at our Haskell representation of polynomials, you will easily convince yourself that the number of valid polynomials of a given degree d equals the number of valid numbers that can be presented in the numeral system base q with $d + 1$ digits. If, for instance, $q = 2$, then we have (without the zero-polynomial P [0])

degree	size	polynomials
0	1	$P\ [1]$
1	2	$P\ [0,1], P\ [1,1]$
2	4	$P\ [0,0,1], P\ [1,0,1], P\ [0,1,1], P\ [1,1,1]$
3	8	$P\ [0,0,1,1], P\ [1,0,1,1], P\ [0,1,1,1], P\ [1,1,1,1]$ $P\ [0,0,0,1], P\ [1,0,0,1], P\ [0,1,0,1], P\ [1,1,0,1]$
...	...	...

We, hence, can precisely say how many polynomials of degree $< d$ there are, namely $r = q^d$. For the example $q = 2$, we see that there are 16 polynomials with degree less than 4, which is 2^4. One of those polynomials, however, is $P\ [0]$, which we must exclude, when asking for $\varphi(g)$, since, for this polynomial, division is not defined. For the irreducible polynomial g, we therefore have $r-1$ elements that do not share divisors with g, *i.e.* $\varphi(g) = r - 1$. So, according to Euler's theorem, we have

$$f^{r-1} \equiv 1 \pmod{g}. \tag{8.25}$$

Multiplying both sides by f, we get

$$f^r \equiv f \pmod{g}. \tag{8.26}$$

Since $r = q^d$, this is equivalent to 8.24 and this concludes the proof. □

From Fermat's theorem, we can derive a nice and useful corollary. Note that, when we subtract f from both sides of the equivalence, we would get 0 on the right-hand side, which means that g divides the expression on the left-hand side. Set $x = f$, then we have:

$$x^{q^d} - x \equiv 0 \pmod{g}. \tag{8.27}$$

This is the basis for a nice test for irreducibility. Since the group established by a non-irreducible polynomial of degree d has less than $p^d - 1$ elements, it will divide $x^{p^c} - x$ for some $c < d$, but an irreducible polynomial will not. Here is a Haskell implementation:

```
irreducible :: Natural → Poly Natural → Bool
irreducible p u  | d < 2       = False
                 | otherwise = go 1 x
  where d       = degree u
        x       = P [0,1]
        go i z = let z′ = powmp p p z
                 in case pmmod p (addp p z′ (P [0,p − 1])) u of
                   P [_]  → i ≡ d
                   g      → if i < d then go (i + 1) (pmmod p z′ u)
                                     else False
```

The function receives two arguments: the modulus and the polynomial we want to check. First, we compute the degree of the polynomial. When the polynomial is of degree 0 or 1, there are by definition only trivial, *i.e.* constant factors. It is, hence, not irreducible (it is not reducible either, it is just uninteresting). Then we start the algorithm beginning with values 1 and x, where x is the simple polynomial x. In *go*, we raise this polynomial to the power of p, and subtract it from the result. Note that we add $p - 1$, which, in modular arithmetic, is the same as subtracting 1. We take the result modulo the input polynomial u. This corresponds to $x^{p^d} - x$ for degree $d = 1$.

If the result is a constant polynomial and the degree counter i equals d, then equation 8.27 is fulfilled. (Note that we consider any constant polynomial as zero, since a constant polynomial is just the content, which usually should have been removed before we start to search factors.) Otherwise, if the degree counter does not equal d, this polynomial fulfils the equation with a “wrong” degree. This is possible only if the input was not irreducible in the first place.

Finally, if we have a remainder that is not constant, we either continue (if we have not yet reached the degree in question) or, if we had already reached the final degree, we return with False, since the polynomial is certainly not irreducible.

Note that we continue with *pmmod p z′ u*, that is, with the previous power modulo u. This is an important optimisation measure. If we did not do that, we would create gigantic polynomials. Imagine a polynomial of degree 8 modulo 11. To check that polynomial we would need to raise x to the power of 11^8, which would result in a polynomial of degree 214 358 881. Since the only thing we want to know is a value modulo u, we can reduce the overhead of taking powers by taking them modulo u in the first place.

Let us look at an example. We generate a random polynomial of degree 3 modulo 7:

```
g ← randomPoly 7 4
```

I get the polynomial $P\ [3, 3, 3, 4]$. (Note that you may get another one!) Calling *irreducible* 7 *g* says: *False*.

When we raise the polynomial $P\ [0,1]$ to the power of $7^3 = 343$, we get a polynomial of degree 343 with the leading coefficient 1. When we subtract $P\ [0,1]$ from it, it will have -1, which is 6 in this case, as last but one coefficient. Taking this modulo to the random polynomial g, we get the polynomial $P\ [0,3,6]$, which is $6x^2 + 3x$ and certainly not constant. g is therefore not irreducible.

Let us try another one:

```
g ← randomPoly 7 4
```

This time, I get $P\ [3,1,4,4]$. Calling *irreducible* 7 g says: *True*. When we take $x^{7^3} - x$ modulo g, we get $P\ [0]$. But we do not get a constant polynomial for $x^7 - x$ or $x^{7^2} - x$. $P\ [3,1,4,4]$, hence, is irreducible.

The formula, however, is not only interesting for testing irreducibility. What the formula states is in fact that all irreducible polynomials of degree d are factors of $x^{q^d} - x$. Whenever we construct this expression, we have created the product of all irreducible polynomials of degree d. The irreducible factors of the polynomial we want to factor are part of this product and we can get them out just by asking for the greatest common divisor of $x^{p^d} - x$ and the polynomial we want to factor. This would give us the product of all factors of our polynomial of a given degree.

We need to add one more qualification however. Since we are searching for a *unique* factorisation, we should make sure that we always make the polynomial *monic*, that is, we should remove the leading coefficient by dividing all coefficients by it. This corresponds to content-and-primitive-part factorisation as already discussed above, but in the case of modular arithmetic it is much simpler. Whatever the leading coefficients is, we can just multiply all coefficients by its inverse without worrying about coefficients becoming fractions. Here is an implementation:

```
monicp :: Integer → Poly Integer → Poly Integer
monicp p u = let cs = coeffs u
                 k  = last cs `M.inverse` p
             in P (map (modmul p k) cs)
```

The following function, obtains the products of the factors of a given (monic) polynomial degree by degree. Note that we give the result back as a monic polynomial again. Each result is a tuple of the degree and the corresponding factor product.

```
ddfac :: Natural → Poly Natural → [(Int, Poly Natural)]
ddfac p u   = go 1 u (P [0, 1])
  where n = degree u
        go d v x | degree v ⩽ 0 = [ ]
                 | otherwise     =
                  let x'      = powmp p p x
                      t       = addp p x' (P [0, p − 1])
                      g       = gcdmp p t v
                      (v', _) = divmp p v g
                      r       = (d, monicp p g)
                  in case g of
                     P [_] →     go (d + 1) v' (pmmod p x' u)
                     _     → r : go (d + 1) v' (pmmod p x' u)
```

The real work is done by function *go*. It starts with degree $d = 1$, the polynomial u we want to factor and, again, the simple polynomial x. We then raise x to the power p^1 for the first degree, subtract x from the result and compute the *gcd*. If the result is a constant polynomial, there are no non-trivial factors of this degree and we continue. Otherwise, we store the result with the degree, making g monic.

We continue with the next degree, $d + 1$, the quotient of the polynomial we started with and the factor product g we obtained and the power of x' reduced to the modulo u. The latter is again an optimisation. The former, however, is essential to avoid generating the same factor product over and over again. By dividing the input polynomial by g, we make sure that the factors we have already found are taken out. This works only if the polynomial is squarefree of course. (You might remember the discussion of squarefree numbers in the context of Euler's theorem where we found that, if n is squarefree, then $\varphi(n) = \prod_{p|n} p - 1$, *i.e.* the totient number of n is the product of the primes in the factorisation of n all reduced by 1.) We need to come back to this topic and, for the moment, make sure that we only apply polynomials that are squarefree and monic.

We try *ddfac* on the 4-degree polynomial $u(x) = x^4 + x^3 + 3x^2 + 4x + 5$ modulo 7 and call *ddfac* 7 u and obtain the result

$[(1, P\ [2,4,1]), (2, P\ [6,4,1])]$,

i.e. the factor product x^2+4x+2 for degree 1 and the factor product x^2+4x+6 for degree 2. First, we make sure that these are really factors of u by calling *divmp* 7 ($P\ [2,4,1]$), which shows

$(P\ [6,4,1], P\ [0])$.

We can conclude that these are indeed all the factors of u. But, obviously, $P\ [2,4,1]$ or $x^2 + 4x + 2$ is not irreducible, since it is a second-degree polynomial, but it was obtained for the irreducible factors of degree 1. $P\ [6,4,1]$, on the other hand, was obtained for degree 2 and is itself of degree 2. We can therefore assume that it is already irreducible, but let us check: *irreducible* 7 ($P\ [6,4,1]$), indeed, yields *True*.

But what about the other one? How can we get the irreducible factors out of that one? Here Cantor and Zassenhaus come in. They proposed a simple algorithm with the following logic. We, again, use the magic polynomial $x^{p^d} - x$, but choose a specific polynomial for x, say t. We already have that chunk of irreducible polynomials hidden in (P [2, 4, 1]), let us call it u, and know that those polynomials are factors of both, $t^{p^d} - t$ and u. The approach of Cantor and Zassenhaus is to split the factors so that the problem reduces significantly. We can split t into three parts using the equality

$$t^{p^d} - t = t(t^{(p^d-1)/2} + 1)(t^{(p^d-1)/2} - 1). \tag{8.28}$$

Make sure that the equality holds by multiplying the right-hand side out. By a careful choice of t, we can make sure that the factors are likely to be more or less equally distributed among the latter two factors. That, indeed, would reduce the problem significantly.

Since u and $t^{p^d} - t$ share factors, we can transform the equality into the following variant:

$$u = \gcd(u, t) \times \gcd(u, (t^{(p^d-1)/2} + 1)) \times \gcd(u, ((t^{(p^d-1)/2} - 1)) \tag{8.29}$$

A reasonable choice for t is a polynomial of degree $2d - 1$. With high probability, the factors are equally distributed among the latter two factors of the equation and we indeed reduce the problem significantly. To do so, we compute one of the gcds and continue splitting this gcd and the quotient of u and the gcd further. Should we be unlucky (the gcd contains either no or all of the factors), we just try again with another choice for t. After some tries (less than three according to common wisdom), we will hit a common factor.

There is an issue, however, for $p = 2$. Because in that case, $t^{(p^d-1)/2} - 1 = t^{(p^d-1)/2} + 1$. Consider, to illustrate that, a polynomial modulo 2, for instance P [0, 1, 1] and $d = 3$. Then we have

$$(p^d - 1)/2 = (2^3 - 1)/2 = 7/2 = 3.$$

We raise the polynomial to the power of 3 and get [0, 0, 0, 1, 1, 1, 1]. When we add P [1], we get [1, 0, 0, 1, 1, 1, 1]. But what do we subtract? Let us try *modp* 2 (P [−1]). We get back P [1]. Adding and subtracting 1 is just the same thing here.

But that would mean that our formula would be much poorer. We would not have three different factors, but only two, namely t and $t^{(p^d-1)/2} + 1$. Unfortunately, it is very likely that all the factors end up in the second one and with this, we would not simplify the problem.

The fact that we are now working modulo 2 may help. We first observe that, modulo 2, there is no difference between the polynomials $t^{2^d} - t$ (the magic one with $p = 2$) and $t^{2^d} + t$. The second one, however, is easy to split, when we set

$$w = t + t^2 + t^4 + \dots + t^{2^{d-1}}.$$

Then, w^2 would be

$$t^2 + t^4 + \dots + t^{2d}.$$

This may shock you on the first sight. But remember, we are still working modulo p and we have (*freshman's dream*):

$$(a + b)^p \equiv a^p + b^p \pmod{p}.$$

When multiplying w by itself, we would get

$$t^2 + 2t^3 + t^4 + 2t^5 + 2t^6 + t^8.$$

Since we are working modulo 2, all terms with even coefficients cancel out, we, hence, get

$$t^2 + t^4 + t^8.$$

Now, observe that

$$w^2 + w = t^2 + t^4 + \dots + t^{2^d} + t + t^2 + \dots + t^{2^{d-1}},$$

when we rearrange according to exponents, we again get pairs of equal terms:

$$w^2 + w = t + 2t^2 + 2t^4 + \dots + 2t^{2^{d-1}} + t^{2^d}.$$

When we compute this modulo 2, again all terms with even coefficients fall away and we finally get

$$w^2 + w = t^{2^d} + t. \tag{8.30}$$

The point of all this is that we can split the expression $w^2 + w$ into two more or less equal parts, just by factoring w out: $w(w + 1)$. Now, it is again very probable that we find common divisors in both of the factors, w or $w + 1$ making it likely that we can reduce the problem by taking the gcd with one of them. Here is an implementation of the Cantor-Zassenhaus algorithm:

```
cz :: Natural → Int → Poly Natural → IO [Poly Natural]
cz p d u | n ⩽ d       = return [monicp p u]
         | otherwise = do
  x ← monicp p < $ > randomPoly p (2 ∗ d)
  let t    | p ≡ 2       = addsquares (d − 1) p x u
           | otherwise = addp p (powmodp p m x u) (P [p − 1])
  let r = gcdmp p u t
  if degree r ≡ 0 ∨ degree r ≡ n then cz p d u
    else do r1 ← cz p d r
            r2 ← cz p d (fst $ divmp p u r)
            return (r1 ++ r2)
  where n = degree u
        m = (p ↑ d − 1) ‘div‘ 2
```

The function receives a natural number, that is the modulus p, an *Int*, d, for the degree, and the polynomial u, the factor product, which we both obtained from *ddfac*. When the degree is equal or greater than n, the degree of u, we are done: we already have a factor of the predicted degree. Otherwise, we generate a random monic polynomial of degree $2d - 1$. Note that, since *randomPoly* expects the number of coefficients, which is $d + 1$, we just pass $2d$.

Then we calculate t. If p is 2, we use *addsquares*, at which we will look in a moment. Otherwise, we raise the random polynomial to the power of $(p^d - 1)/2$ and subtract 1. That is the third factor of equation 8.29. We compute the gcd and, if the result has either degree 0 (no factor was found) or the same degree as u (all factors are in this one), we just try again with another random polynomial. Otherwise, we continue with the gcd and the quotient $u/\gcd$.

Let us try this for the result $(1, P\ [2,4,1])$ we obtained earlier from applying *ddfac* on $P\ [5,4,3,1,1]$. *cz* 7 1 $(P\ [2,4,1])$ gives

$[P\ [6,1], P\ [5,1]]$,

two irreducible polynomials of degree 1. The complete factorisation of $P\ [5,4,3,1,1]$, hence, is

$[P\ [6,1], P\ [5,1], P\ [6,4,1]]$,

which we can test by calling *prodp* (*mulmp* 7) $[P\ [6,1], P\ [5,1], P\ [6,4,1]]$ and we, indeed, get $P\ [5,4,3,1,1]$ back.

For the case where $p = 2$, we use the function addsquares:

```
addsquares :: Int → Natural → Poly Natural → Poly Natural → Poly Natural
addsquares i p x u = go i x x
  where go 0 w _ = w
        go k w t = let t'  = pmmod p (powmp p p t) u
                       w' = addp p w t'
                   in go (k − 1) w' t'
```

which just computes w as $t + t^2 + t^4 + \dots t^{2^{d-1}}$.

Let us try *ddfac* and *cz* with a polynomial modulo 2, *e.g.* $P\ [0,1,1,1,0,0,1,1,1]$, which is of degree 8 and is squarefree (and, per definition, monic). The call *ddfac* $2\ (P\ [0,1,1,1,0,0,1,1,1])$ gives us three chunks of factors:

$[(1, P\ [0,1,1]), (2, P\ [1,1,1]), (4, P\ [1,1,1,1,1])]$.

We see at once that the second and third polynomials are already irreducible, since they already have the specified degree. The first one, however, is of degree 2, but shall contain factors of degree 1. So, let us see what *cz* $2\ 1\ (P\ [0,1,1])$ will yield:

$[P\ [0,1], P\ [1,1]]$.

The complete factorisation of $P\ [0,1,1,1,0,0,1,1,1]$, hence, is

$[P\ [0,1], P\ [1,1], P\ [1,1,1], P\ [1,1,1,1,1]]$.

Now, we still have to solve the problem of polynomials containing squared factors, *i.e.* repeated roots. There is in fact a method to find such factors adopted from calculus and, again, related to the derivative. The point is that a polynomial and its derivative share only those divisors that appear more than once in the factorisation. We have not enough knowledge on derivatives yet to prove that here rigorously, but we can give an intuition.

Consider a polynomial with the factorisation

$$(x + a)(x + b) \dots$$

This is a product and, to find the derivative of this polynomial, we need to apply the *product rule* (which we will study in part 3). The product rule states that

$$(f \times g)' = fg' + f'g, \tag{8.31}$$

i.e. the derivative of the product of f and g is the sum of the product of f and the derivative of g and the product of the derivative of f and g.

The derivatives of the individual factors $(x+a)(x+b)$ all reduce to 1, since for $f = x^1 + a$, $f' = 1 \times x^0 = 1$. The product of factors, hence, turns into a sum of factors:

$$1 \times (x+a) + 1 \times (x+b) = (x+a) + (x+b).$$

Let us compute the polynomial with the factors $(x+a)(x+b)$. The polynomial is $x^2 + ax + bx + ab$. Its derivative is $2x + a + b$. When we apply the product rule to the factors, we get $(x+a)+(x+b) = x+a+x+b = 2x+a+b$, which is indeed the same result.

It is intuitively clear that the sum of the factors is not the same as the product of those same factors and, even further, that none of the factors is preserved. They all disappear in favour of others they do not share divisors with, because, since the factors are coprime to each other, they do not share divisors with their sum either.

To elaborate on this, consider now polynomials with more than two factors of the form

$$abc\dots,$$

where a, b and c stand for irreducible polynomials like $(x+\alpha)$, $(x+\beta)$ and so on.

We apply the product rule on the first two factors and get:

$$(a'b + ab')\dots$$

When we now apply the product rule once again, we would multiply c with the derivative of ab (which is $a'b + ab'$) and the derivative of c, c', with the original ab and get:

$$(a'b + ab')c + abc' = a'bc + ab'c + abc'.$$

We see that we end up with the sum of the products of the original factors, with the current factor i substituted by something else, namely the derivative of this factor. This can be represented nicely as:

$$\left(\prod_{i=0}^{k} a_i\right)' = \sum_{i=0}^{k}\left(a_i' \prod_{j \neq i} a_j\right) \tag{8.32}$$

There is a remarkable similarity to the structure we found in analysing the Chinese remainder theorem, when we divided the product of all remainders by the current remainder. Just as in the Chinese remainder theorem, each of the terms resulting from the product rule is coprime to the original factor at the same position, since it is the product of irreducible factors (which, hence, are coprime to each other) and the derivative of that factor, which, for sure, does not share divisors with the original factor at that position.

When we have a repeated factor, however, as in the following polynomial

$$(x+a)(x+a)(x+b)\dots,$$

then this factor is preserved. The product rule will create the factor $x+a+x+a = 2x+2a$, which is a multiple of the original factor, which, in its turn, is therefore preserved.

Suppose we want to compute the factorisation of

$$f = a_1 a_2^2 a_3^3 \dots a_k^k, \tag{8.33}$$

where the as represent the products of all the factors raised to the indicated exponent, then, since the derivative preserves the factors, the gcd of f and its derivative f' (whatever that looks like) is:

$$\gcd(f, f') = a_2^1 a_3^2 \dots a_k^{k-1}, \tag{8.34}$$

i.e. the repeated factors with the exponent decreased by one. Then f divided by the gcd gives us

$$\frac{f}{\gcd(f, f')} = a_1 a_2 a_3 \dots a_k, \tag{8.35}$$

all the factors reduced to their first power. Now, if we continue this scheme using the $\gcd(f, f')$ and $f/\gcd(f, f')$ as input, we would get 8.34 reduced one more $(a_3 a_4^2 \dots)$ and 8.35 with the head chopped off $(a_2 a_3 \dots)$. The quotient of the two versions of 8.35, *i.e.*

$$\frac{a_1 a_2 a_3 \dots}{a_2 a_3 \dots},$$

would give us the head. The head, however, is the product of all factors with a given exponent.

In a finite field, this, unfortunately does not work in all cases. Problematic are all coefficients with exponents that are multiples of the modulus. When we compute nc^{n-1}, for n an exponent in the original polynomial that is a multiple of the modulus, the coefficient itself becomes zero. If we are unlucky, the derivative *disappears*, *i.e.* it becomes zero. A simple example is the polynomial $x^4 \pmod 2$. When we compute the derivative, we get $4x^3$. Unfortunately, 4 is a multiple of 2 and, therefore, the only nonzero coefficient we had in the original polynomial becomes zero and the entire derivative disappears.

What we can do, however, is to keep the coefficients with exponents that are multiples of the modulus separated from those that are not. As in the algorithm for infinte fields, we would iteratively compute two sequences of values, namely $T_{k+1} = T_k/V_{k+1}$ with $T_1 = \gcd(f, f')$ and $V_{k+1} = \gcd(T_k, V_k)$ with $V_1 = f/T_1$. But we would now deviate for all k that are multiples of p, *viz.*

$$V_{k+1} = \begin{cases} \gcd(T_k, V_k) & \textit{if } p \nmid k \text{ (as before)} \\ V_k & \textit{if } p \mid k \end{cases}$$

At each step, we have

$$V_k = \prod_{i \geq k, p \nmid i} a_i, \tag{8.36}$$

i.e., the product of all as with exponents greater than those that we have already processed and that do not divide p, and

$$T_k = \prod_{i \geq k, p \nmid i} a_i^{i-k} \prod_{i \geq k, p \mid i} a_i^i, \tag{8.37}$$

i.e., the product of the powers greater than those we have already processed for both cases $p \mid i$ and $p \nmid i$. For the cases $p \nmid i$, everything is as before. For the cases $p \mid i$, we will end up, when we have reduced V_k to a constant polynomial, with a product of all the powers of the as with exponents that are multiples of p.

To get these as out, we divide all all exponents by p and repeat the whole algorithm. For the return value, *i.e.* the factors, we need to remember the original exponent, but that is easily done as shown below.

Note that for polynomials with many coefficients, this recursion step will occur more than once. The exponents that are multiples of p in such a polynomial have the form

$$0p, p, 2p, 3p, 4p, \dots$$

Dividing by p, we get

$$0, 1, p, 2p, 3p, \dots$$

So, we need to repeat, until there are no more multiples of p. Here is the algorithm:

```
sqmp :: Integer → Integer → Poly Integer → [(Integer, Poly Integer)]
sqmp p e u | degree u < 1  = [ ]
           | otherwise     = let u' = derivative (modmul p) u
                                 t  = gcdmp p u u'
                                 v  = fst (divmp p u t)
                             in go 1 t v
  where go k tk vk = let vk' | k ‘rem‘ p ≢ 0 = gcdmp p tk vk
                             | otherwise = vk
                         tk' = fst (divmp p tk vk')
                         k'  = k + 1
                     in case divmp p vk vk' of
                        (P [_], _) →                    nextStep k' tk' vk'
                        (f, _)     → (k ∗ p ↑ e, f) : nextStep k' tk' vk'
        nextStep k tk vk | degree vk > 0 = go k tk vk
                         | degree tk > 0 = sqmp p (e + 1) (dividedTk tk)
                         | otherwise     = [ ]
        dividedTk tk = poly (divExp 0 (coeffs tk))
        divExp _ [ ] = [ ]
        divExp i (c : cs) | i ‘rem‘ p ≡ 0 = c : divExp (i + 1) cs
                          | otherwise     =     divExp (i + 1) cs
```

As usual, the hard work is done in the local function *go*, which takes three arguments, k, t_k and v_k. We initialise $k = 1$, $t_k = \gcd(u, u')$ and $v_k = u/t_k$. We set $v_{k+1} = \gcd(t_k, v_k)$, if $p \nmid k$, and $v_{k+1} = v_k$, otherwise. We further set $t_{k+1} = t_k/v_{k+1}$ and $k = k + 1$. If v_k/v_{k+1} (this is the head) is not constant (otherwise it is irrelevant), we remember the result as the product of factors with this exponent. Note that the overall result is a list of tuples, where the first element represents the exponent and the second the factor product. The exponent is calculated as $k \times p^e$. The number e, here, is not the Euler-Napier constant, but a variable passed in to *sqmp*. We would start the algorithm with $e = 0$. We, hence, get $k \times p^0 = k \times 1 = k$ for the first recursion.

The function *nextStep* is just a convenient wrapper for the decision of how to continue. If v_k is not yet constant, we continue with *go* $(k + 1)$ t_{k+1} v_{k+1}. Otherwise, if t_k is not yet constant, we continue with *sqmp* with $e+1$ and t_k with exponents that are multiples of p divided by p.

For bootstrapping the algorithm, we can define a simple function with a reasonable name that calls *sqmp* with $e = 0$:

```
squarefactormod :: Integer → Poly Integer → [(Integer, Poly Integer)]
squarefactormod p = sqmp p 0
```

Finally, we are ready to put everything together:

```
cantorzassenhaus :: Integer → Poly Integer → IO [(Integer, Poly Integer)]
cantorzassenhaus p u | irreducible p m = return [(1, m)]
                     | otherwise       =
                       concat < $ > mapM mexpcz [(e, ddfac p f) |
                                          (e, f) ← squarefactormod p m]
  where m = monicp p u
        expcz e (d, v)   = map (λf → (e, f)) < $ > cz p d v
        mexpcz (e, dds) = concat < $ > mapM (expcz e) dds
```

Hans Zassenhaus worked most of his life as a computeralgeabrist and was important for the development of this area of mathematics and computer science. He was born in Germany before the second world war and studied mathematics under Emil Artin, one of the founders of modern algebra. Zassenhaus' father was strongly influenced by Albert Schweitzer and, as such, opposed to Nazi ideology. Hans shared this antipathy and, to avoid being drafted to a significant war effort like, as it would appear natural for an algebraist, cryptography, he left university and volonteered for the army weather forecast where he survived the war. Later, he would follow invitations first to the UK and later to the USA, where he remained until his death.

His sister Hiltgunt (who, after emigrating to the USA, preferred to use her second name Margret) studied Scandinavistics. During the war, she worked as translator for censorship in camps for Norwegian and Danish prisoners. She undermined censorship in this position, maintained contact between prisioners and helped smuggling medicine, tobacco and food into the prisons. For her efforts during and after the war, she was nominated for the Nobel Peace Prize in 1974. Unfortunately, the societal engagement of the Zassenhaus siblings is hardly remembered today.

8.10. Hensel Lifting

8.11. The closed Form of the Fibonacci Sequence

$$G(x) = F_0 + F_1x + F_2x^2 + F_3x^3 + \dots \tag{8.38}$$

$$G(x) = 0 + x + x^2 + 2x^3 + 3x^4 + 5x^5 + 8x^6 + \dots \tag{8.39}$$

$$xG(x) = F_0x + F_1x^2 + F_2x^3 + F_3x^4 + \dots \tag{8.40}$$

$$x^2G(x) = F_0x^2 + F_1x^3 + F_2x^4 + F_3x^5 + \dots \tag{8.41}$$

$$G(x) - xG(x) - x^2G(x) = (1 - x - x^2)G(x). \tag{8.42}$$

$$\begin{array}{rllllll}
(1 - x - x^2)G(x) = & (F_0 & +F_1x & +F_2x^2 & +F_3x^3 & + \dots) & - \\
& (& F_0x & +F_1x^2 & +F_2x^3 & + \dots) & - \\
& (& & +F_0x^2 & +F_1x^3 & + \dots) &
\end{array}$$

$$\begin{aligned}
(1 - x - x^2)G(x) = \quad & F_0 + (F_1 - F_0)x \\
& + (F_2 - F_1 - F_0)x^2 \\
& + (F_3 - F_2 - F_1)x^3 \\
& + \dots
\end{aligned}$$

$$(1 - x - x^2)G(x) = x. \tag{8.43}$$

$$G(x) = \frac{x}{1 - x - x^2}. \tag{8.44}$$

9. Relations, Functions and the Cartesian Plane

9.1. Relations

9.2. Equivalence Classes

9.3. Functions

9.4. Plotting Functions

9.5. Slopes and Intersects

9.6. Trigonometry

9.7. Navigation

9.8. Conic Sections

9.9. Algebra and Geometry

10. Linear Algebra

10.1. Vectors

10.2. Vector Spaces

10.3. Clustering

10.4. Linear Maps and Operators

10.5. The Matrix

10.6. Eigenvalues

10.7. Inner Products and their Operators

10.8. Polynomials and Vector Spaces

10.9. Determinants

10.10. Support Vector Machines

11. Elliptic Curves

11.1. Geometry Intuition

Before we go on with more theoretical topics, let us examine another application of the quite abstract theory of algebra we have studied in the previous chapters, namely elliptic curve cryptography. It should be mentioned that linear algebra is by far not the only topic relevant in this context. In fact, important aspects of the theory of elliptic curves require the understanding of function analysis – where it actually comes from – but here we will focus on algebra and group theory.

Elliptic Curves (EC) provide the mathematical background for variants of public key cryptography. This kind of cryptography is being developed since the eighties, but it took a while until it was accepted by the industry. Today, however, it is the main public key cryptography scheme around. Its acceptance was accelerated by the smartphone boom. In smartphones and other devices with restricited resources, classic cryptographic schemes are not very practical. Their drawback is the computational overhead resulting from key size. Cryptoanalytic attacks forced classic schemes to be used with huge keys. To achieve 128-bit security with RSA, for instance, we need keys with at least 3072 bits. The same level of security can be reached with EC cryptography, according to known attacks today, with 256 bits. A huge improvement!

EC cryptography is different from classic cryptography in various respects. First, it includes much more math. That is to say, it does not include theory from only one or two branches of mathematics like number theory in classic cryptography, but from many different branches. This has huge impact on cryptoanalysis. Hidden attacks may lurk in apparently remote fields of mathematics that we did not account for. However, the theory surrounding EC is very well understood today and, as said, it is the mainline cryptography approach today.

Second, the basic means, especially the group we need for public key cryptography, are much more "engineered" than in classic cryptography. Classic schemes are based mainly on modular arithmetic, which was well known centuries before anyone thought of this use case. The groups found in modular arithmetic, in particular the multiplicative group, was then used to define cryptographic tools. In elliptic curves, there are no such groups "by nature". They are constructed on the curves with the purpose to use them in cryptography. Therefore, EC may sometimes feel a bit artificial. It is important to understand that the group we define on the curves is defined voluntarily according to our purpose. When we speak of *point addition* in this context, one must not confuse this operation with the arithmetic operation of addition. It is something totally different.

Anyway, what are elliptic curves in the first place? Elliptic curves are polynomials that were intensively studied in the late 19^{th} century, especially by German mathematician Karl Weierstrass (1815 – 1897), who was of huge importance in the sound fundamentation of analysis. We will meet him again in the third part. He studied polynomials of the form

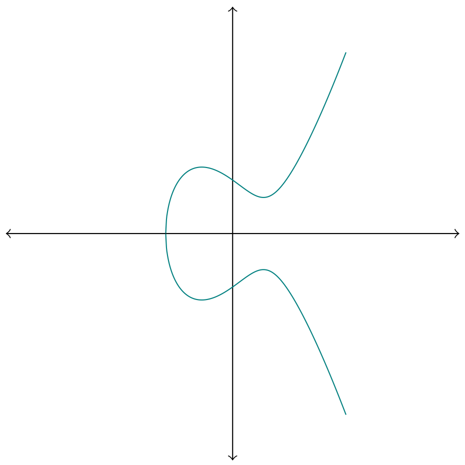
$$y^2 = x^3 + ax + b, \tag{11.1}$$

which is said to be in *Weierstrass form*. We can easily transform this equation into a form that looks more like something that can be computed, namely:

$$y = \sqrt{x^3 + ax + b}. \tag{11.2}$$

But be careful! Weierstrass polynomials are not functions, at least not in $\mathbb{R}$, since there is not exactly one y for each x. When the expression $x^3 + ax + b$ becomes negative, there is, in the world of real numbers, no solution for the right-hand side of the equation.

This is quite obvious, when we look at the geometric interpretation of that polynomial. It looks – more or less – like in the following sketch:

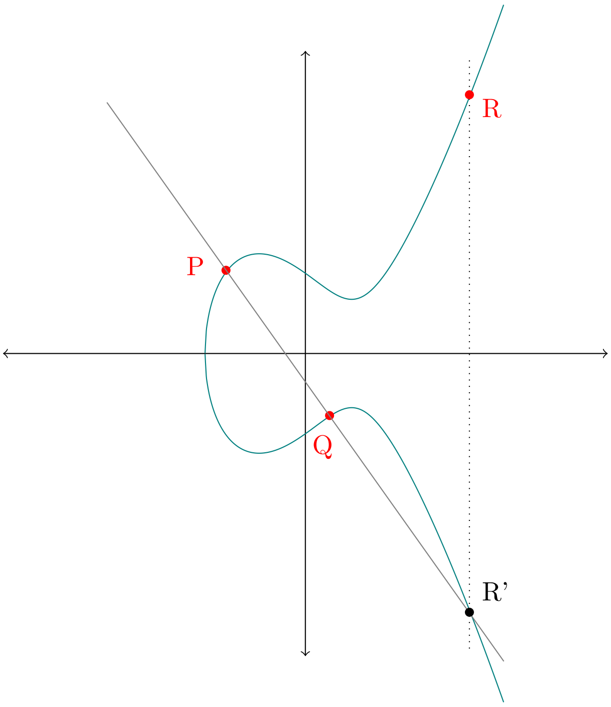


The exact shape depends on the coefficients a and b. The bubble on the left may sometimes be a circle or ellipse completely separated from the "tail" on the right; it may, in other cases, be less clearly distinguished from the tail on the right, forming just a tiny bulge in the tail.

In any case, the curve “ends” on the left-hand side for some $x < 0$. More precisely, it ends where the absolute value of x^3, for a negative value, becomes greater than $ax + b$. Then, the whole expression becomes negative and no real square root corresponds to it.

We will now start to construct a group on this kind of curves. We call it an *additional group*, but be aware that this is not addition in the sense of the arithmetic operation. It has nothing to do with that! It is a way to combine points with each other that can be captured in a – more or less – simple formula. We will start by giving a geometric interpretation of this operation. This will help getting an intuition. But, again, be aware that we are not dealing with geometry. We will soon deviate from geometry and talk about curves in a quite abstract way.

The following sketch shows an elliptic curve with three points P, Q and R, all coloured in red. These points are in the relation $P + Q = R$.



When adding two points P and Q on an elliptic curve, we draw a straight line through them (the grey one). From the nature of the elliptic curve, it is obvious that the straight line will meet the curve once again. At that intersection, we draw a helper point, R'. Then we reflect this point across the x-axis, *i.e.* we draw another line (the dotted one) that goes straight up crossing R'. This line will meet the curve again, namely at a point with the same x coordinate, but with the inverse of the y coordinate $-y$. That point is R, the result of $P + Q$.

You see that this operation has in fact nothing to do with arithmetic addition. It is an arbitrary construction to relate three points. Nevertheless, it is carefully designed to give rise to a group based on this operation, as we will see later.

For the moment, our main question is how can we compute R from P and Q. We start by computing the straight line. A straight line is defined by a formula of the form

$$y = mx + c, \tag{11.3}$$

where m is the slope and c the y-intercept. What we need to do now is to find the third point, R', which, like P and Q, lies on both, the straight line and the elliptic curve. To find such a point, we set the two formulas equal. Since an elliptic curve is defined as

$$y^2 = x^3 + ax + b, \tag{11.4}$$

we can say

$$(mx + c)^2 = x^3 + ax + b. \tag{11.5}$$

By subtracting $(mx + c)^2$ from both sides, we get

$$x^3 + ax + b - (mx + c)^2 = 0. \tag{11.6}$$

Using the binomial theorem we can expand this to

$$x^3 - m^2x^2 - 2mxc - c^2 + ax + b = 0. \tag{11.7}$$

We already know two points, where this equation is fulfilled, namely x_P and x_Q. This means that these values are roots of the above equation. We can hence use them for factoring that equation into $(x - x_P)(x - x_Q)\Psi$, where Ψ is yet another factor. But we know even more. We just have to look at the sketch above to see that there are three roots and, hence, three factors. We, therefore, have $\Psi = x - x_{R'}$ and conclude that

$$x^3 - m^2x^2 - 2mxc - c^2 + ax + b = (x - x_P)(x - x_Q)(x - x_{R'}). \tag{11.8}$$

From here it is quite simple. We just apply the trick of the *opposite sum of the roots* and get

$$m^2 = x_P + x_Q + x_{R'}, \tag{11.9}$$

which we can easily transform to

$$x_{R'} = m^2 - x_P - x_Q. \tag{11.10}$$

Since R, the point we are finally looking for, is the reflection of R' across the x-axis, we have $x_R = x_{R'}$, *i.e.* the points have the same x-coordinate.

Computing $y_{R'}$ is again quite simple. The points P and R' are on the same straight line. The y-values on a straight line increase at a constant rate. So, the value of y should grow travelling on the segment between x_P and x_R, which is $m(x_R - x_P)$ and add this to the already known y-value at point P:

$$y_{R'} = y_P + m(x_R - x_P). \tag{11.11}$$

Now we compute y_R, the y-coordinate of the reflections of R' across the x-axis, which is simply $-y$. Alternatively, we can compute that value directly by rearranging the equation to

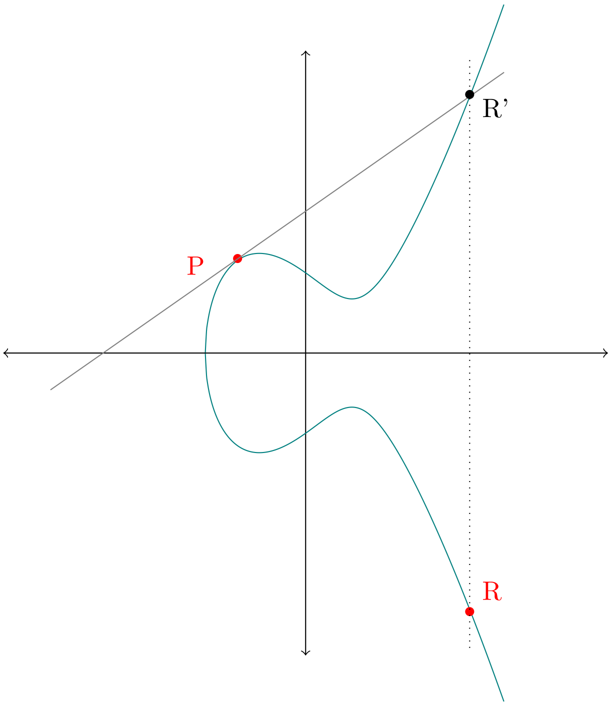
$$y_R = m(x_P - x_R) - y_P. \tag{11.12}$$

The final piece missing now is the slope, m, which can be expressed as a fraction:

$$m = \frac{y_Q - y_P}{x_Q - x_P}. \tag{11.13}$$

With this equation, however, we get into trouble. Everything is fine, when we assume that we add two distinct points P and Q. But if we have $P = Q$, *i.e.* if we want to add a point to itself, then the denominator of the above fraction becomes negative. That, clearly, is to be avoided.

To avoid that, we use, instead of a secant line that intersects the curve, the tangent line at point P, which, as we already know, measures the slope of the curve at P. Geometrically, this corresponds to the following sketch:



Here, we draw the tangent line at P. Where the tangent line intersects the curve again, we draw the helper point R'. We reflect it across the x-axis and obtain the point $R = P + P = 2P$.

As you hopefully remember, the slope of a curve at a given point can be calculated with the derivative of that curve. We will apply that derivative trick to get the tangent line at P. This task, however, is a bit more difficult than for the trivial cases we have seen so far. Until now, we have seen derivatives of simple functions like $f(x) = x^2$, whose derivative is $f'(x) = 2x$. Now, we have the equation

$$y^2 = x^3 + ax + b. \tag{11.14}$$

We can interpret this equation as an application of two different functions. The first function, say g, is $g(x) = x^3 + ax + b$. The second function, f, is $f(x) = \sqrt{x} = x^{\frac{1}{2}}$.

For such cases, we have the *chain rule*, which we will discuss more thoroughly in part 3. The chain rule states that the derivative of the composition of two functions is

$$(f \circ g)' = (f' \circ g) \times g'. \tag{11.15}$$

That is, the derivative of the composition of two functions f and g is the derivative of f applied on g times the derivative of g. Let us figure out what the derivatives of our f and g are. The derivative of g is easy:

$$g'(x) = 3x^2 + a$$

A bit more difficult is f'. If $f(x) = x^{\frac{1}{2}}$, then

$$f'(x) = \frac{1}{2}x^{\frac{1}{2}-1} = \frac{1}{2}x^{-\frac{1}{2}} = \frac{1}{2x^{\frac{1}{2}}}.$$

Now, we apply this to the result of $g(x)$, which we can elegantly present as y^2. If we plug y^2 into the equation above, we get

$$\frac{1}{2y^{2\times\frac{1}{2}}} = \frac{1}{2y}.$$

We now multipy this by g' and get

$$\frac{3x^2 + a}{2y}.$$

When we use this formula for $x = x_P$, we get the formula to compute m:

$$m = \frac{3x_P^2 + a}{2y_P}. \tag{11.16}$$

So, we finally have an addition formula that covers both cases, $P \neq Q$ and $P = Q$:

$$x_R = \begin{cases} m^2 - x_P - x_Q & \text{if } x_P \neq x_Q \\ m^2 - 2x_P & \text{otherwise} \end{cases} \tag{11.17}$$

and

$$y_R = m(x_P - x_R) - y_P, \tag{11.18}$$

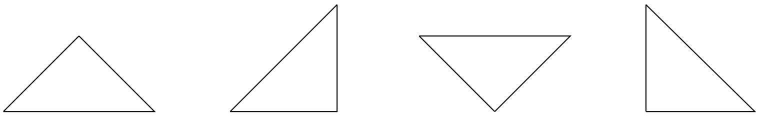
where

$$m = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} & \text{if } x_P \neq x_Q \\ \frac{3x_P^2 + a}{2y_P} & \text{otherwise.} \end{cases} \tag{11.19}$$

11.2. Projective Geometry

To complete the construction of the group of points on an elliptic curve, we still have to define the identity and the inverse. To do this we make a detour through the beautiful art of *projective geometry.* To be honest, we do not need too many concepts of projective geometry in practice. But with an intuitive understanding of those concepts the jargon common in EC cryptography becomes much clearer. Besides, projective geometry is really a beautiful part of mathematics worth studying whether we need it for EC or not.

Geometry is often concerned with difference and equality of quantities like length and angle. In this type of geometry, called *metric geometry*, one studies properties of objects under transformations that do not change the length and angle. A typical statement is, for instance, that two triangles are congruent (and hence equal), when one of them can be seen as a roation or displacement of the other. The triangles below, for instance, are all congruent to each other:

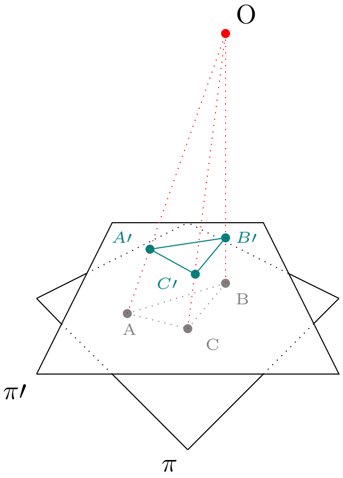


In fact, we could say it is four times the same triangle tumbling around. We are now looking at what remains from metric geometry, when we disregard length and angle as first-class properties of objects. One way to do so is by looking at logic configurations according to some basic notion, such as that of parallel lines. This gives rise to what is called *affine geometry.* Another way is to look at properties invariant under projective transformations and this is indeed what projective geometry does.

The triangles above are, as you can see, drawn on a plane. We could now take another plane, just as we would grab a piece of paper, and *project* the points on the first plane onto the second plane. The two planes do not need to fulfil any specific configuration. They may be parallel to each other or they may not. They may be arranged in any configuration relative to some orientation in the surrounding space. Indeed, we are now looking at transformations of two-dimensional figures on two-dimensional planes. But the transformations are created by projecting one figure through three-dimensional space onto another plane. Of course, we can generalise this to n-dimensional planes in an $n + 1$-dimensional space. But that would be far beyond our needs.

Projective transformations relate points on one plane, let us call it π, to points on the other plane, $\pi\prime$, by drawing a straight line that relates both points with yet another point, which, in *central projection*, is called the centre and is identical for all points we project from π to $\pi\prime$. We can also choose to use *parallel projection*, where points are projected by parallel lines, each one having its own projection "centre". As we will see later, in projective geometry, that is not a significant difference. The latter is just a special case of the first resulting from a very specific choice of the central point.

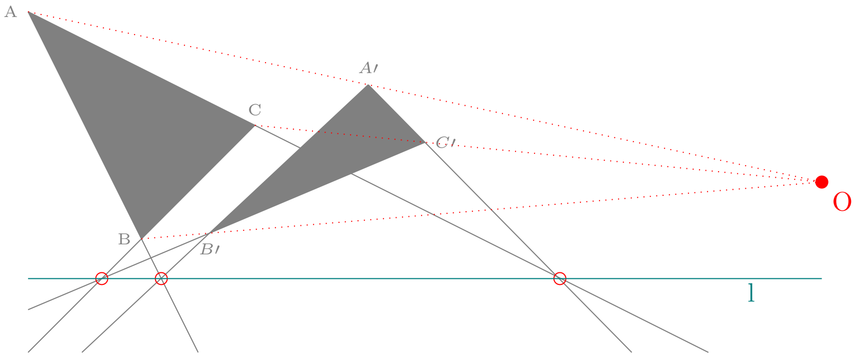
The following sketch shows a projection from π, the lower plane, to $\pi\prime$ using central projection with O being the central point:



There are, on π, three points, A, B and C, which form a triangle. The points are projected on $\pi\prime$ along the lines relating each of the points with O. We see that all points on π appear on $\pi\prime$ and we see that one of the properties that are preserved is that on both planes these points form a triangle pointing roughly in the same direction. The triangles, however, are of different size and, due to different arrangement of the planes in space, the shape of the original triangle is distorted on $\pi\prime$.

Projective geometry studies properties that remain unchanged under projection. Such properties are essential for us recognising projected shapes. It is therefore no surprise that projective geometry was originally introduced to mathematics by math-literate painters, in particular Renaissance artists like Leonardo da Vinci (1452 – 1519) and Albrecht Dürer (1471 – 1528). Today projective geometry is ubiquitous. It is used in all kinds of image processing and image recognition. It is widely used in digital cameras for instance, but also in many other kinds of applications.

The founding father of the mathematical discipline of projective geometry was the French engineer, architect and mathematician Girard Desargues (1591 – 1661). Desargues formulated and proved *Desargues' theorem*, one of the first triumphs of projective geometry. The theorem states that, if two triangles are situated such that the straight lines joining corresponding vertices of the triangles intersect in a point O, then the corresponding sides, when extended, will intersect in three points that are all on the same line. Here is a sketch to make that a bit clearer:



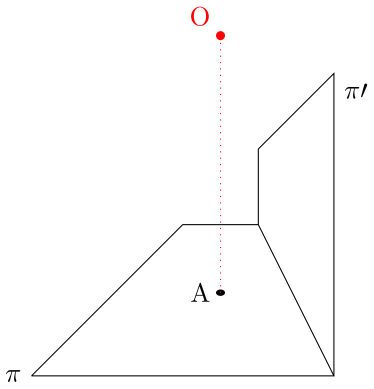
The dotted lines capture the theorem's precondition: corresponding vertices of the triangles lie on lines that intersect in one point O. The gray lines extend the sides of the triangles and the pairs of corresponding sides all intersect, each side with its corresponding side, on the same line l.

The theorem looks quite simple; after all, it contains only straight lines. It is nevertheless quite difficult to prove with means of metric geometry. If we consider the two triangles being on different planes, however, and one the projection of the other, the argument suddenly becomes very easy.

We first note that all points and lines making up one triangle are located on one plane. We then observe that each of the lines that relate one edge of one triangle with one edge of the other triangle, for instance $\overline{AA\prime}$ or $\overline{BB\prime}$, also lie in a plane, otherwise we could not draw these lines. But that means that A and $A\prime$, B and $B\prime$ and C and $C\prime$ as well as O all lie in the same plane. Therefore the lines $\overline{AB}$ and $\overline{A\prime B\prime}$ must meet somewhere. Since the triangles are in separate planes, the two planes must meet somewhere too and there, where the planes meet, there must be the intersection of all those lines. Two planes, however, meet in a line and, since they have exactly one line in common, it must be on that line where all the other lines itersect.

Note that this proof works with reasoning according to the logic of plane and space alone, which makes it concise and elegant, but also quite subtle. Indeed, I hesitate to put "□" to the end of the proof. In fact, there is a flaw in it. The proof only works when the planes are not parallel to each other! When we project the triangle onto a plane parallel to the first one, then these two planes will certainly never meet – and that crashes the proof.

That is a very typical situation in projective geometry. Theorems and proofs would look very nice and clean, had we not always those exceptions of parallel lines! In fact, there are even points on the original plane that will never appear in the projection, because their projective line is parallel to the second plane. Point A in the following configuration, for instance, with the projective centre at O will never show up on the target plane:



Projective geometry could be very clean and nice, was there not that issue of parallel lines. It comes into the way in every axiom and every theorem and every proof. Therefore, mathematicians tried to come around it. They did so by the following thought experiment. If we have two intersecting lines and now start to rotate them slowly so that they approximate the configuration where they are parallel to each other, the point of intersection moves farther away towards infinity. We could then assume that all lines intersect. There is then nothing special about parallel lines. They intersect too, but do so very far away, *viz.* at infinity. This way, we extend the concept of point and line by adding one point to each line, namely the point where this line and all lines parallel to it intersect. That point is then said *to be at infinity.*

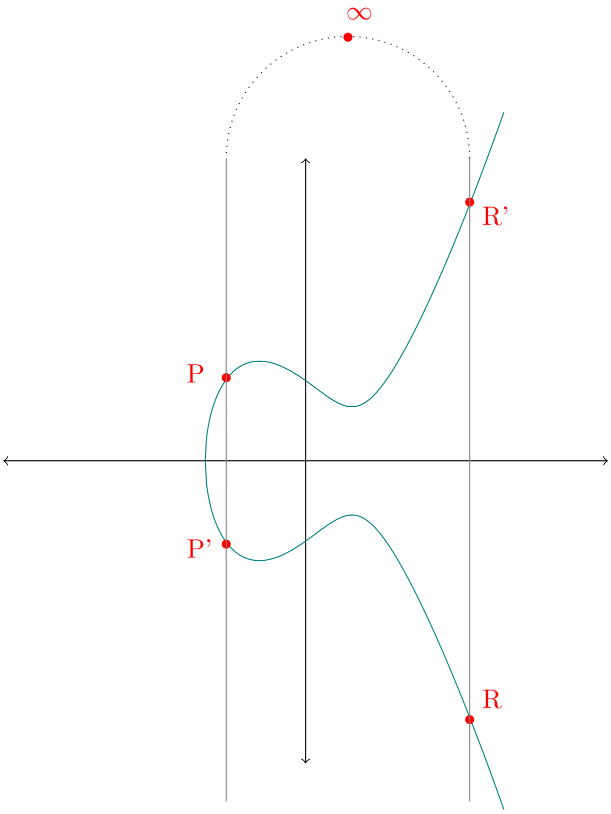
This trick to extend a concept is very similar to how we extended natural numbers to integers by adding a sign. Suddenly, we had a solution for problems that were unsolvable before, namely subracting a number from a smaller one.

Out there, at infinity, there are now many points, each one the intersection of an infinite number of parallel lines crossing it. Of course, we now can draw a line through all these points at infinity, the *line at infinity.* That is where the planes in the proof of Desargues' theorem intersect in the case where they are parallel to each other. We then only have to prove that, if one pair of lines intersect at infinity, the other two as well intersect at infinity.

It is perhaps worth to emphasise that this is not the result of observation of physical reality. Nobody has ever seen two lines intersecting at infinity. It is not a statement about physics at all. It is an axiom that we assume, because it makes reasoning in projective geometry much easier.

This handling of parallelism separates the two approaches to geometry, affine and projective geometry. Indeed, in affine geometry the notion of parallel lines is central. Many problems in that branch of mathematics are centred on the implications of one line being parallel to another or not. In projective geometry, by contrast, two lines being parallel to each other is nothing special. It just means that their point of intersection is very far away. We will discuss this again later in more detail.

Let us now come back to elliptic curves. Where do parallel lines play a role in our addition formula? Well, the line to reflect a point across the x-axis is parallel to the y-axis and all such lines are parallel to each other. In projective terminology, all these lines intersect at infinity. In other words, a point and its reflection define a line that intersects with the reflection lines of all other points at the point at infinity. We can sketch that like this:



Usually, when we add two points on an elliptic curve, we search for a third intersection of the straight line through the points with the curve and then reflect that point across the x-axis. What should happen, when we do this with a point and its reflection across the x-axis? It is indeed not quite clear, because we will not find any other intersection of line and curve. However, if we continue to travel along the line, we would at some point ("at infinity") reach the intersection of the line we are travelling with all other lines parallel to the y-axis. The idea now is to define addition of a point P with its reflection P' in such a way that $P + P'$ is precisely that point, which, in the context of elliptic curves, we call $\mathcal{O}$. So we add that point $\mathcal{O}$ to the curve and decide – deliberately – that this point is the additive identity. For any point P on the curve, it then holds that $P + \mathcal{O} = P$. The point P', for which $P + P' = \mathcal{O}$, *i.e.* the reflection of P across the x-axis, is in consequence the inverse of P.

Note that there is no deeper mathematics involved here that would directly lead to a formula that we could apply to "automatically" generate the result $P+P' = \mathcal{O}$. Instead, we have to consider this case as well as $P + \mathcal{O} = P$ explicitly in the addition formula.

But why do we reflect at all, when adding two points? That is because, otherwise, addition would be quite boring. Suppose we added without reflection. Then addition would go $P + Q = R'$ and the reverse additions $R' + Q = P$ and $R' + P = Q$ would just lead back to where we started. This would be true for any three points in such a constellation, because the three points are on the same straight line. If we go forward prolonging the line $\overline{PQ}$, we find R'. If we go backward prolonging the line $\overline{R'Q}$, we find P, or, if we draw the line $\overline{R'P}$, we find Q in the middle. Even if such a rule could ever lead to a group, it would not be cyclic, *i.e.* there would be no generators. A generator in elliptic curve cryptography is a point that repeatedly added to itself creates the whole group. But leaving reflection out, the subsequent addition of a point P would give raise to a sequence like $P, 2P, P, 2P, P, \dots$, which, certainly, is not a group.

11.3. EC modulo a Prime

It was already indicated that the geometry exercises in the previous sections had the sole purpose of giving an intuition. EC Cryptography does not take place in the continuous universe. It does take place in modular arithmetic with integers and, hence, in a discrete world. This is a disruptive turning point, since we cannot plot a curve and search for a point in the Cartesian plane anymore. As we will see examining points of a curve modulo some number these points are not located on anything even close to the curves we saw in the previous sections. One could say that we adopt the algebra of elliptic curves, but drop the geometry.

Let us start with a data type. We define an elliptic curve as

```
data Curve = Curve {
  curA :: Natural,
  curB :: Natural,
  curM :: Natural}
  deriving (Show, Eq)
```

This type describes a curve in terms of its coefficient a and b and in terms of the modulus. When we consider only curves of the form

$$y^2 = x^3 + ax + b, \tag{11.20}$$

the definition given by the type is sufficient. There are other curves, though, for instance this one:

$$y^2 = x^3 + ax^2 + bx + c, \tag{11.21}$$

but we do not consider them in this humble introduction.

For the modulus, either a (huge) prime is used or a (huge) power of 2. Again, we do not consider powers of 2.

Now we define the notion of "point":

```
data Point = O | P Natural Natural
  deriving (Eq)
```

and make it an instance of *Show* to get a more pleasant visualiation:

```
instance Show Point where
  show O = "O"
  show (P x y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Note that we explicitly define $\mathcal{O}$, the identity, to which we will have to refer explicitly in addition and other operations on points later.

We also define convenience getters for the point coordinates:

```
xco :: Point → Natural
xco O = error "O has no coordinates"
xco (P x _) = x

yco :: Point → Natural
yco O = error "O has no coordinates"
yco (P _ y) = y
```

To be sure that the points we create are modulo p, we define a convenient creator function:

```
point :: Curve → (Natural, Natural) → Point
point c (x, y) = P (x `mod` p) (y `mod` p)
  where p = curM c
```

Note that we use our *mod* function defined in section on modular arithmetic in the Prime chapter.

Now we would like to have a function that gives us the y-coordinate of the point with a given x-coordinate. In the continuous universe that would be quite easy. It is a bit complicated in modular arithmetic. We start with a function that gives us y^2:

```
curveY'2 :: Curve → Natural → Natural
curveY'2 c x = (x ↑ 3 + a * x + b) `mod` p
  where a = curA c
        b = curB c
        p = curM c
```

That is neat and simple. We just plug the given x-value into the right-hand side of the curve equation and get y^2 back. But, now, how to compute y? In the continuous universe, we would just call $\sqrt{y^2}$. But we are in modular arithmetic and y^2 is not necessarily a perfect square, but a quadratic residue, which may or may not be a perfect square. Here are as an example the residues of prime 17:

$$0, 1, 2, 4, 8, 9, 13, 15, 16.$$

Those are nine numbers, which was to be expected, since, for any prime modulus p, there are $\frac{p+1}{2}$ residues and $\frac{p-1}{2}$ nonresidues. Of these nine numbers, only five, namely 0, 1, 4, 9 and 16, are perfect squares. For those it is quite easy to compute the root. It is just the regular square root. For the others, however, it is quite hard. The problem is closely related to the Discrete Logarithm Problem (DLP), which is hard enough to provide the setting for most public key cryptographic schemes around today. Anyway, we have to live with it for the moment and implement a searching algorithm that is fine for small modulus, but infeasible in practice:

```
findRoot :: Natural → Natural → Natural
findRoot p q = go 0
  where go x | x > p       = error "not found!"
             | (x ↑ 2) `mod` p ≡ q = x
             | otherwise = go (x + 1)
```

Basically, we just go through all numbers from 0 to $p-1$, until we find one that squared yields q, the residue in question. If we do not find such a number, we terminate with an error. If we map *findRoot* on the residues of 17, *map* (*findRoot* 17) (*residues* 17), we see:

$$0, 1, 6, 2, 5, 3, 8, 7, 4.$$

Some numbers are not surprising at all. 0 is of course the root of 0 and so is 1 of 1, 2 of 4, 3 of 9 and 4 of 16. But who had thought that 6 is the root of 2, 8 that of 13 or 7 that of 15?

With the help of this root finder, we can now implement a function that gives us y for x:

```
curveY :: Curve → Natural → Maybe Natural
curveY c x = let r = curveY'2 c x
             in if isSqrM r p then Just (findRoot p r)
                              else Nothing
  where p = curM c
```

We have inserted a safety belt in this function. Before we go into *findRoot*, which may cause an error when there is no root for the number in question, we check if it is a residue at all. If it is, we are confident to find a root and just return the result of *findRoot*. Otherwise, we return *Nothing*, meaning that the curve is not defined for this specific x. Here is the test for r being a residue using the Legendre symbol:

```
isSqrM :: Natural → Natural → Bool
isSqrM 0 _ = True
isSqrM n p = legendre n p ≡ 1
```

Based on these functions, we can define other useful tools. A function that verifies wether a given point is on the curve:

```
oncurve :: Curve → Point → Bool
oncurve _ O         = True
oncurve c (P x y) = case curveY c x of
                      Nothing → False
                      Just z    → y ≡ z ∨ y ≡ p − z
  where p = curM c
```

The function receives a curve and a point. It determines the y-coordinate for the x-coordinate of the point. If no y-coordinate is found, the point is certainly not on the curve. Otherwise, if the y we found is the same as the one of the point, then the point is on the curve. If the value we found is $-y$, that is to say, $p-y$, then the point is also on the curve, because $p-y$ is the additive inverse of y in the group and, if the point $(x,-y)$ is on the curve, then (x,y), the inverse of the point, is also in on the curve. Note that, when we say "a point is on the curve", we effectively say "the point is in the group". But be careful: we are here referring to two different groups. The group of integers modulo p and the group of points that "are on the curve".

The function *oncurve* is not very efficient, since it needs the root to calculate the result of *curveY*. A more efficient version is this one:

```
oncurve′2 :: Curve → Point → Bool
oncurve′2 _ O         = True
oncurve′2 c (P x y) = let z = curveY′2 c x
                       in  (y ↑ 2)        ‘mod‘ p ≡ z ∨
                           (p − y) ↑ 2 ‘mod‘ p ≡ z
  where p = curM c
```

As we are already dealing with inverses, here are two functions, one finding the inverse of a point and the other testing if a point is the inverse of the other:

```
pinverse :: Curve → Point → Point
pinverse _ O         = O
pinverse c (P x y) = point c (x, −y)

isInverse :: Curve → Point → Point → Bool
isInverse _ O O = True
isInverse c p q   = q ≡ pinverse c p
```

Another useful tool would be one that finds us a point on the curve. There are two ways to do it: deterministic and random. We start with the deterministic function that would basically go through all number from 0 to $p-1$ and stop, whenever there is a y for this x, such that (x,y) is on the curve:

```
findPoint :: Curve → Point
findPoint c = let (x, y′) = hf [(x, curveY′2 c x) | x ← [1 ..]]
              in  point c (x, findRoot p y′)
  where hf  = head ∘ filter (ism p ∘ snd)
        p   = curM c
        ism = flip isSqrM
```

The function generates tuples of the form (x,y^2) and filters those where y^2 is indeed a residue of p. The first of the resulting list is returned and laziness saves us from going through literally all possible x. This is a very useful tool to get started with a curve, but it is a bit boring, because it would always yield the same point. Randomness would make that more exciting giving us different points. Here is a function that yields a random point on a given curve:

```
randomPoint :: Curve → IO Point
randomPoint c = do
  x ← randomRIO (1, p − 1)
  let y′ = curveY′2 c x
  if isSqrM y′ p then return (point c (x, findRoot p y′))
                 else randomPoint c
  where p = curM c
```

The code is straight forward. First we generate a random number x in the range $1 \dots p-1$. Then we determine y^2 and, if this is a residue, we return the point consisting of x and y. Otherwise, if it is not a residue, we start all over again.

Let us take a break here and look at some points in a real curve. We start by defining a curve for experiments:

```
c1 :: Curve
c1 = Curve 2 2 17
```

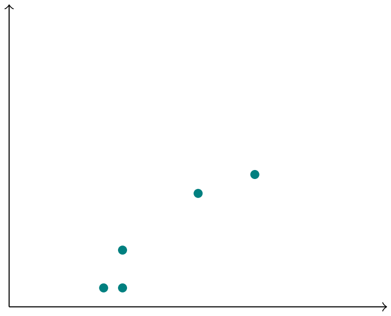
This corresponds to the curve

$$y^2 \equiv x^3 + 2x + 2 \pmod{17}.$$

We call `mapM (\_ → randomPoint c1) [1 ..5]`, generating five random points. We may see the points

$$(10,6),(5,1),(6,3),(3,1),(13,7)$$

(or any other selection of points. It is a **random** list!) As expected, we see points with integer coordinates in the range $0 \dots 16$. Let us look where those points are located in the Cartesian plane.



As already said: that does not look like an elliptic curve at all. It does not look completely random either. To have the complete picture, however, we need all points on that curve. How can we get them? Right! With a generator! Where do we get a generator? One way is trial and error. But for that we need the group operation. So let us get on with it. Here is addition:

```
add :: Curve → Point → Point → Point
add _ _ p O = p
add _ _ O p = p
add c p@(P x1 y1) q@(P x2 y2) | isInverse c p q = O
                              | otherwise        =
                                let xr = (l ↑ 2 − x1 − x2)     ‘mod‘ m
                                    yr = (l ∗ (xr − x1) + y1) ‘mod‘ m
                                in point c (xr, −yr)
  where a             = curA c
        m             = curM c
        l | x1 ≡ x2   =
          let t1      = (3 ∗ x1 ↑ 2 + a)   ‘mod‘ m
              t2      = inverse ((2 ∗ y1) ‘mod‘ m) m
          in  (t1 ∗ t2) ‘mod‘ m
          | otherwise =
          let t1      =           (y2 − y1) ‘mod‘ m
              t2      = inverse ((x2 − x1) ‘mod‘ m) m
          in  (t1 ∗ t2) ‘mod‘ m
```

We start with the base cases where one of the points is $\mathcal{O}$, the identity of the group of the curve. The result of addition in this case is just the other point. Then we handle two points none of which is the identity. If one is the inverse of the other, then the result is just $\mathcal{O}$. All these cases, as already mentioned in the previous section, must be explicitly handled in our implementation. There is no direct way that would produce the result. After all, this is a highly "engineered" group.

Now, we are finally in the "regular" case, where none of the points is the identity and the points are not the inverses of each other. In this case – we just apply the formula we have learnt before. However, it looks a bit different. This is because we are now in the discrete universe of modular arithmetic. The main difference is that, instead of dividing coordinates, we multiply them by the modular inverse of the denominator. We are here dealing with the group of integers modulo the prime we use for the curve.

It should be mentioned that to compute the slope of the line l, we distinguish the cases $p = q$ (point doubling) and $p \neq q$ by just comparing the x-coordinates ignoring the y-coordinates. We can do this, because we already have checked one point being the inverse of the other. Since the inverse of a point (x, y) is its reflection across the x-axis $(x, -y)$ and there, for sure, is no other point with that x-coordinate, it would be redundant to check the y-coordinate once again.

What do points look like, when we add them up? Let us take two points from the list above. What about the first two, $(10, 6)$ and $(5, 1)$? We add them calling *add c1* (P 10 6) (P 5 1) and get

$$(3, 1).$$

There is really nothing that would suggest any similarity to ordinary arithmetic addition.

How can we use addition to generate the whole group? Since we are dealing with an additive group (according to this strange definition of addition), we can pick a primitive element, a generator, and add it successivley to itself. But what is a primitive element of the group of our curve $c1$? Well, I happen to know that the order of that group is 19. Since we are talking about groups, Lagrange's theorem applies, *i.e.* the order of subgroups must divide the order of the main group. Therefore, all members of the group are either member of a trivial subgroup (which contains only one element, namely the identity) or generators of the main group. Since the sole element in the trivial group is the identity $\mathcal{O}$, all other members of the group must be generators. We, hence, can pick any point and generate the whole group from it. Here is a generator function:

```
gen :: Curve → Point → [Point]
gen c p = go p
  where go O = [O]
        go r = r : go (add c r p)
```

We call it like *gen c1* (P 10 6) and get

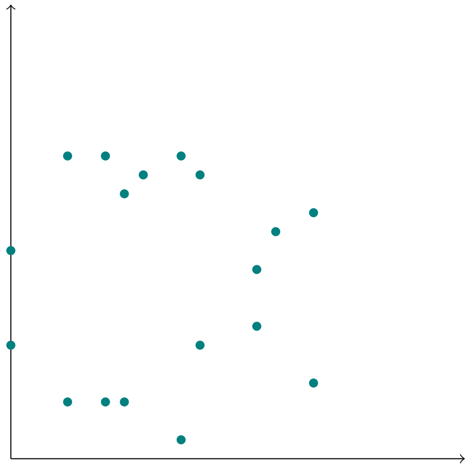
$$(10,6),(16,13),(7,6),(0,11),(3,16),(5,16),(6,3),$$
$$(9,16),(13,7),(13,10),(9,1),(6,14),(5,1),(3,1),$$
$$(0,6),(7,11),(16,4),(10,11),O,$$

which are 19 points and, hence, the entire group of the curve $c1$.

Note that the final point is the identity. This is exactly the same behaviour as we saw for multiplicative groups modulo a prime. For instance, 3 is a generator of the group modulo 7. We saw that $3^1 \equiv 3$, $3^2 \equiv 2$, $3^3 \equiv 6$, $3^4 \equiv 4$, $3^5 \equiv 5$ and $3^6 \equiv 1$ all $\pmod{7}$.

The last but one point in the list is the inverse of the point we started with. In the integer case, there was nothing obvious that pointed to the fact that 5 is the inverse of 3 modulo 7. With the points above, however, it is immediately clear, since, as you can see, the penultimate point is $(10, 11)$. It has the same x-coordinate as $(10, 6)$ and the y-coordinate is $-y$ of the original point, because $17 - 6 = 11$. 11, hence, is -6 modulo 17.

Do we get a clearer picture when we put all these points on the Cartesian plane? Not really:

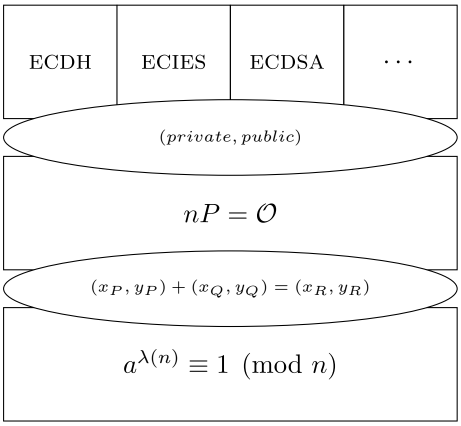


So, let us forget about geometry for a while. We are dealing with modular arithmetic related to a construction that we happen to call a curve. There is no more secret geometry behind it.

11.4. The EC Discrete Logarithm Problem

To summarise the results from the previous sections, we can describe EC Crypto as a system with three layers. The first layer consists in modular integer arithmetic and is used to do the math underlying point addition. Point addition itself belongs to the second layer, the additive group of points on the curve, which results from *using* point addition. Since all arithmetic modulo a prime is done within point addition, when we look at point addition itself, we do not “see” modular arithmetic anymore. We do not add points modulo something, we just add them using the addition formula.

The third layer consists of the cryptosystems build on top of the group of points. To actually build cryptosystems on top of elliptic curves, we need a secret that we can use as private key and an open parameter acting as public key. Perhaps the following schematic sketch helps:


ECDH
ECIES
ECDSA
$\cdots$
$(private, public)$
$nP = \mathcal{O}$
$(x_P, y_P) + (x_Q, y_Q) = (x_R, y_R)$
$a^{\lambda(n)} \equiv 1 \pmod{n}$

Bottom up, we see first the arithmetic group modulo some prime, indicated by Carmichael's theorem. On top of it, we built point addition, which, in its turn, is the basis for the group of points on the curve, indicated by the fact that repeated addition (*i.e.* multiplication) of a point P yields the identity of that group $\mathcal{O}$. By means of this group, we build a key pair (*public*, *private*), which then enables us to build concrete cryptographic schemes, such as EC Diffie-Hellman (ECDH), EC Integrated Encryption Scheme (ECIES) and the EC Digital Signature Algorithm (ECDSA), which we will discuss in the next sections. As indicated by the right-most box in that layer, there are many more cryptographic schemes for elliptic curves. But we will not discuss all of them here.

When we look back at classic cryptography, we see that we typically used an invertible operation, namely *exponentiation*. In some cases, like in Diffie-Hellman, the private key was a combination of public keys of the form $a^{xy} = (a^x)^y = (a^y)^x$, in others, like RSA, public and private keys were inverses of each other, such that $a^{ed} = a^1 = a$. In either case, the security of the private key was based on the hardness of the discrete logarithm problem (DLP). The DLP aims to solve equations of the form

$$a^x = b$$

for x, *i.e.* it asks for the number x, to which we have to raise a to get to b. The classic crypto schemes are based on a multiplicative group. The DLP, hence, asks for the number of repetitions of successive multiplications of a to get to b. In EC, however, we have an additive group or, at least, we use "additive terminology" describing the group operation. When we ask for the number of repetitions of that operation on a to get b, we would hence ask for multiplication, not for exponentiation. There is some potential for confusion in this terminology switch from multiplicative to additive groups. The problem on which the security in EC relies is unfortunately also called DLP, discrete logarithm problem. It would be much more precise in this case to refer to the *discrete quotient problem*, since we talk about multiplication, not exponentiation. However, the terminology is like that. So, we stick to it and define the DLP for EC crypto.

The information of EC crypto systems that is publicly known consists of parameters describing the curve, the coefficients, the modulus, a starting point P and perhaps some other details specifying the exact curve. The public key is typically a point Q and the secret is a natural number n, such that

$$nP = Q. \tag{11.22}$$

The DLP, hence, consists in finding a *factor* n that determines how often we have to add P to itself to get to Q. This may sound weird, but, in fact, it is a hard problem equivalent in computational complexity to finding the discrete logarithm in classic cryptography.

Consider the curve we already used above with starting point $P = (5, 1)$. If you consider the public key to be $Q = (9, 16)$, can you tell, without looking at the whole group above, what n must be, such that $n \times (5, 1) = (9, 16)$? The point is that no algorithm is known that would do that in acceptable time for large groups. For this toy group, we obviously can try. We would see that:

$$\begin{aligned}
(5, 1) + (5, 1) &= (6, 3) = 2P,\\
(6, 3) + (5, 1) &= (10, 6) = 3P,\\
(10, 6) + (5, 1) &= (3, 1) = 4P,\\
(3, 1) + (5, 1) &= (9, 16) = 5P.
\end{aligned}$$

We, thus, find our point after four additions, which corresponds to multiplying the point by 5. With a group that contains 2^{256} elements, this approach would not be feasible.

But how can it then be feasible to compute nP for large n in the first place? We would need n steps to produce that result and n can be a really large number. Obviously, we need a way to perform multiplication in significantly less than n steps, in $\log n$ steps, for instance.

There is indeed a way to do this. The algorithm is quite similar to the algorithm we used to raise a number to a huge power, which was *multiply-and-square*. Here we use a variant of that algorithm called *double-and-add*, since we are dealing with an additive group. The algorithm may be implemented like this:

```
mul :: Curve → Natural → Point → Point
mul _ 0 _  = O
mul _ _ O = O
mul c n p  = foldl da p (tail $ toBinary n)
  where da q 0 = add c q q
        da q 1 = add c p (add c q q)
```

The first base case states that 0 times anything is the identity. Then, the identity multiplied by any number is again the identity. The identity, indeed, never changes with multiplication. Just as zero would never change by multiplication in the realm of numbers. Any point multiplied by zero, on the other hand, is the identity of the additive group and that, again, is zero.

In all other cases, we convert n into a list of binary digits of which we process all but the head (of which we know it is 1, otherwise it would not be the head). For each digit, we double the intermediate result q, that is we compute *add c q q*. If the current digit is 0, we are done with this digit and continue with the next one. Otherwise, if it is 1, we additionally add p. When there are no more digits left, we have a result.

It is noteworthy that this implementation of multiplication does not use the means of the arithmetic group modulo a prime that underlies point addition. It is build on top of addition using only terms related to the group of points on the curve, the second layer in the sketch above.

Let us look at an example. Say, we want to compute $19P$. The binary representation of 19 is $[1,0,0,1,1]$. We, hence, would compute $P+P$ for the first digit 0 (which is the head of the tail of our number). This is $2P$. With this result we go into the next round. The next digit is 0 again and we get $4P$, which is the input for the next iteration. The next digit is 1, so we double and add. We, hence, compute $4P+4P+P=9P$. This is now the input to the final digit. Since it is 1 again, we again double and add and we have $9P+9P+P=19P$.

It works perfectly. But why does it work? Consider the representation of a number in terms of powers of 2 multiplied by a number k (corresponding to our point P above):

$$(a_r 2^r + a_{r-1} 2^{r-1} + \dots + a_0 2^0)k,$$

where, for $i \in \{0 \dots r\}$, $a_i \in \{0,1\}$. Multiplying this out, we get

$$a_r 2^r k + a_{r-1} 2^{r-1} k + \dots + a_0 2^0 k.$$

Obviously, from step to step, that is from plus sign to plus sign, right to left, k doubles. Ignoring the coefficients a_i for a moment, this would look for the concrete number $10011_2 = 19_{10}$ like

$$16k + 8k + 4k + 2k + k.$$

Doubling alone would in this case generate the number $16k$, which would indeed be the correct result if the binary number were 10000_2. Now, we eliminate all terms with coefficient $a_i = 0$, which are $8k$ and $4k$. We are left with

$$16k + 2k + k.$$

The value we add to $16k$ corresponds exactly to the value of k we would add with *mul*. For the example, we would add one k processing the last but one digit. This k is now part of the intermediate result, $2q + k$, that goes into the processing of the last digit. Processing the last digit, we double the previous result, obtaining $4q + 2k$ and, since the last digit is also 1, we add k again. We, hence, get three "extra" ks, which we add to the overall doubling result 16 and get $16 + 2 + 1 = 16 + 3 = 19$.

11.5. EC Diffie-Hellman

Diffie-Hellman on elliptic curves (ECDH) is very similar to Diffie-Hellman on the group of remainders of a prime. We just change the group operation. Everything else remains (more or less) the same. We start by defining the group parameters:

```
data ECDHParams = ECDHP Curve Point
```

That is, the parameters we need to establish the protocol in the first place are the curve we are using and a starting point, which is a generator of the group of the curve. Now we define a function to generate a random private key:

```
ecdhRandomPrivate :: ECDHParams → IO Natural
ecdhRandomPrivate (ECDHP c g) = randomNat (2, o − 1)
  where o = gorder c g
```

In plain English, we select a random number between two and the size of the group minus one. From this we can compute the public key, which we can exchange over the unsecure channel:

```
ecdhPublic :: ECDHParams → Natural → Point
ecdhPublic (ECDHP c g) d = mul c d g
```

The public key, hence, is a point, namely the result of the multiplication of the start point, g, by a number d, which is the number we have just chosen randomly from the range $2 \ldots o - 1$. We put this into a convenient protocol initialisation routine:

```
ecdhInit :: ECDHParams → IO (Natural, Point)
ecdhInit ps@(ECDHP c g) = do
  k ← ecdhRandomPrivate ps
  return (k, ecdhPublic ps k)
```

Before we go on to present the communication between Alice and Bob, we define a concurrent printing function that should help us inspecting what is going on between the two of them:

```
type PFun = String → IO ()
put :: MVar () → String → IO ()
put m s = withMVar m (\_ → putStrLn s)
```

As you can see, the function locks an *MVar* before writing to *stdout*. The intention is to avoid ending up with jumbled strings printed to *stdout*.

We further implement two pairs of functions that together establish a secure channel on top of an unsecure one. The key we are using to secure the channel is the result of the Diffie-Hellman key exchange protocol. Note that these functions are not part of Diffie-Hellman itself. We, in fact, use a quite stupid encryption just for illustration purpose:

```
ecdhEncrypt :: Point → Natural → Point
ecdhEncrypt (P x y) m = P (xor (x + y) m) 0

ecdhDecrypt :: Point → Point → Natural
ecdhDecrypt (P x y) (P m _) = xor (x + y) m
```

The first function encrypts a message (represented as a natural number) using a key, which is a point. The encryption is just an *xor* of the message using the sum of the coordinates of the point. Decryption consists in *xor*ing the cipher, again, with the sum of the point coordinates. (This, certainly, is not a good encryption algorithm – but, finally, this is not a cryptography tutorial!)

Note that the encryption result is a point, not a number. This has no further significance. We just do that, because we want to use the same channels for exchanging messages that were used for establishing the key. Since these channels are defined as *Chan Point*, the cipher sent through this channel must be a point too. Indeed, the y-coordinate of the cipher is just 0. It has no meaning whatsover.

The next pair of functions use the cryptographic functions above to send a message (which is just a natural number) through the otherwise unsecure channel:

```
sfSend :: Chan Point → Point → Natural → IO ()
sfSend ch p m = writeChan ch (ecdhEncrypt p m)

sfRead :: Chan Point → Point → IO Natural
sfRead ch p = (ecdhDecrypt p) < $ > readChan ch
```

We now implement Alice, the one who initiates the protocol:

```
alice :: ECDHParams → PFun → Chan Point → Chan Point → IO ()
alice ps@(ECDHP c g) sfp ich och = do
  (d, qa) ← ecdhInit ps
  sfp ("Alice: private key is " ⧺ show d)
  writeChan och qa
  qb ← readChan ich
  let k = mul c d qb
  sfp ("Alice: common key is " ⧺ show k)
  m ← randomNat (1, o − 1) :: IO Natural
  sfp ("Alice: sending " ⧺ show m)
  sfSend och k m
  threadDelay 1000000
  where o = gorder c g
```

This function receives the *ECDHParams*, the concurrent printing function and two channels, one outgoing and the other incoming. Alice starts by initialising the protocol creating her secret, d and her public key, qa (standing for point q multiplied by alice's secret). Alice prints the secret to *stdout*, so we can follow what is happening.

She then sends the public key, which is a point, through the outgoing channel (*och*) and waits on the incoming channel (*ich*) for Bob's answer. From the answer, she creates the joint secret by multiplying Bob's point (qb) by her own secret d. She prints it to *stdout* for us to see and then creates a random number, which is the message to be protected by the common secret. Note that, due to our simplistic encryption function, the message must be a number in the group we are happening to use. This is not realistic of course.

Alice prints the message to *stdout* and sends it encrypted by the common secret, k, through the unsecure channel (which is just the outgoing channel used before). Finally, we delay the task for a second to give all players the time to finish their business.

Here is what Bob does:

```
bob :: ECDHParams → PFun → Chan Point → Chan Point → IO ()
bob ps@(ECDHP c g) sfp ich och = do
  (d, qb) ← ecdhInit ps
  sfp ("Bob   : private key is " ⧺ show d)
  qa ← readChan ich
  writeChan och qb
  let k = mul c d qa
  sfp ("Bob   : common key is " ⧺ show k)
  m ← sfRead ich k
  sfp ("Bob   : received: " ⧺ show m)
  threadDelay 1000000
```

Bob initialises the protocol, obtaining a secret key and a public key, prints the secret key to *stdout* and waits for Alice to start the protocol. At some point he receives Alice's public key, *qa*, and then sends his own public key, *qb*. He computes the common secret *k* and prints it to *stdout* for us to see. Then he waits for the encrypted message on the now secured channel and writes it to *stdout*. Finally, we delay the task for a second to give all players the time to finish their business.

Of course, we also need the evesdropper Eve:

```
eve :: PFun → Chan Point → Chan Point →
              Chan Point → Chan Point → IO ()
eve sfp aich aoch bich boch = do
  a ← readChan aoch
  sfp ("Eve  : Alice to Bob: " ⧺ show a)
  writeChan bich a
  b ← readChan boch
  sfp ("Eve  : Bob to Alice: " ⧺ show b)
  writeChan aich b
  m ← readChan aoch
  sfp ("Eve  : Alice to Bob: " ⧺ show m)
  writeChan bich m
  threadDelay 1000000
```

Eve is a *woman-in-the-middle* and holds all channels, those used by Alice and those used by Bob. That this is possible means that these channels are unsecure. Alice and Bob cannot rely on them when communicating confidential messages between them, such as love letters that should be kept away from their parents; or explosive news kept from the public by the authoritarian government that rules the country where Alice and Bob live.

Eve reads Alice channel, prints what she sees, and sends the message as it is to Bob. Then she waits for Bob's response. When it arrives, she again prints what she sees and sends it to Alice. She now waits for Alice message, which will be encrypted by our simple *xor*ing encryption function. She again prints what she sees, sends it to Bob and delays execution for a second for the others to terminate their stuff.

The point here is that Eve can see everything that is on the channel. But she cannot guess what the common secret key is Alice and Bob are using. She sees two numbers, the public key of Alice and that of Bob. But she does not know the factors d that were used by them to generate those numbers. To know them, she would need to solve the ECDLP, which is hard for huge numbers. Alice and Bob, on the other hand, compute the shared secret by computing $a \times qb$ and $b \times qa$, respectively. Since qb, Bob's public key, is $b \times q$ and qa, Alice's public key, is $a \times q$, we end up with the computations $a \times b \times q$ and $b \times a \times q$. These computations, however, since group operations are associative, result in the same number.

Here is a demo program that puts all the pieces together:

```
ecdhDemo :: IO ()
ecdhDemo = do
  aich ← newChan
  aoch ← newChan
  bich ← newChan
  boch ← newChan
  m ← newMVar ()
  let sfp = put m
  let ecdhp = ECDHP (Curve 2 2 17) (P 5 1)
  void $ forkIO (eve sfp aich aoch bich boch)
  void $ forkIO (alice ecdhp sfp aich aoch)
  void $ forkIO (bob ecdhp sfp bich boch)
  threadDelay 5000000
```

Since the protocol is not deterministic, but uses random numbers, the output of this program will vary between calls. A possible output is:

```
Alice: private key is 13
Bob  : private key is 2
Eve  : Alice to Bob: (16,4)
Bob  : common key is (0,6)
Eve  : Bob to Alice: (6,3)
Alice: common key is (0,6)
Alice: sending 11
Eve  : Alice to Bob: (13,0)
Bob  : received: 11
```

The first line is Alice to reveal her secret key to us, which is 13. Then Bob tells us his secret, which is 2. Alice now sends her public key to Bob. This message is intercepted by Eve. She sees the point $(16,4)$. Indeed, $13 \times (5,1)$ in the curve *Curve* 2 2 17 is $(16,4)$.

In the next line we see a triumphant Bob, revealing the shared secret $(0,6)$, that he computed as $2 \times (16,4)$. Now Bob sends his public key $(6,3)$ to Alice, which is observed by Eve. Note that $2 \times (5,1)$ is indeed $(6,3)$. In the next line, Alice has computed the shared secret and it is of course equal to Bob's result, since $13 \times (6,3) = (0,6)$.

Alice now uses the shared secret to send the secret message, 11, through the wire. Eve sees the point $(13,0)$. The x-coordinate of this point, 13, is the result of *xor*ing $0 + 6$ and 11, since $6_{10} = 110_2$ and $11_{10} = 1011_2$. When we *xor*, we compute

0	1	1	0
1	0	1	1
1	1	0	1

and $1101_2 = 13_{10}$. As we can see in the last line, Bob has received the correct message 11.

11.6. EC Integrated Encryption Scheme

The EC Integrated Encryption Scheme, ECIES, is much more complex, but also much more complete than the ECDH. Mathematically, however, it is very similar. It differs from ECDH mainly in what is defined on top of the mathematical basis.

In concrete terms, ECIES defines a complete encryption scheme as part of its parameters. Just as ECDH, it defines in its parameters the curve and the primitive element from where we start. Additionally, it defines a public key (which in Diffie-Hellman is computed *ad hoc* as part of the key exchange protocol), an algorithm to enrich the secret that is derived from the public key, so it can be used in a symmetric encryption function, this encryption function as such and its inverse and an authentication scheme. Since we here focus on the mathematical aspects, we simplify this parameter set a bit. We are not interested in the key enrichment and, for the moment, we are not interested in authentication. We therefore present the ECIES parameters in the much simpler form:

```
data ECIESParams = ECIES Curve Point Point CryptoF
```

and define *CryptoF* as

```
type CryptoF = Natural → Natural → Natural
```

It represents an encryption function that receives a key and a message (both natural numbers) and uses the key to transform the message into a third natural number, the ciphertext. The function is assumed to be its own inverse. So, when it receives the key and the ciphertext, it yields the original message. A simple (and, in practical terms, not very robust) example for such a crypto function is again *xor*.

We now have two keys, the public key known to everybody and the private key known only to the owner of the key, just as in traditional RSA. The key generation, hence, is not part of the protocol. Keys are generated in advance and the public key is made available to people that might want to communicate to the owner of the key pair, say Bob.

So, at some point in time, Bob decides he wants to use encryption for part of his communication and, to this end, he creates a key pair:

```
eciesKeyPair :: Curve → Point → IO (Natural, Point)
eciesKeyPair c g = do
  p ← randomNatural (2, o − 1)
  return (p, mul c p g)
  where o = gorder c g
```

The key generation function *eciesKeyPair* receives a part of the parameters, namely the curve and the primitive element, and generates by means of this input a natural number, the private key, and a point, the public key. The key generation itself is just as in ECDH: we generate a random number p in the range of the order of the underlying group and multiply the generator g by this number. The number p is the private key and the resulting point is the public key.

Based on this key generator function, we can define a function that creates us the parameters:

```
eciesMakeParams :: Curve → Point → CryptoF → IO (Natural, ECIESParams)
eciesMakeParams c g f = do
  (p, k) ← eciesKeyPair c g
  return (p, ECIES c g k f)
```

The function receives a curve, a point (the generator) and a crypto function and yields a natural number (the private key) and the parameters. How it does this, is straight forward.

Now, we look at encryption. To encrypt a message to Bob, Alice would use Bob's public key to derive a common secret. This step is very similar to ECDH. The difference is that the public key is known beforehand. Here is a function for Alice to derive a secrete using Bob's public key:

```
eciesSecret :: ECIESParams → IO (Natural, Point)
eciesSecret ps@(ECIES c g k _) = do
  r ← randomNatural (2, o − 1)
  let p = mul c r g
  let q = mul c r k
  if   q ≡ O then eciesSecret ps
             else return (xco q, p)
  where o = gorder c g
```

Alice starts by, again, selecting a random number in the order of the group. She multiplies this number with the generator to obtain point p and multiplies it with Bob's public key to obtain point q. If q is the identity, she tries again. (Note that p cannot be the identity, since r is in the order of the group and g is a primitive element). Otherwise, she returns the x-coordinate of q (the product of r and Bob's public key) and the point p (the product of r and the generator).

Here is how she uses the secret to encrypt a message m:

```
eciesEncrypt :: ECIESParams → Natural → IO (Point, Natural)
eciesEncrypt ps@(ECIES c g k f) m = do
  (s, p) ← eciesSecret ps
  return (p, f s m)
```

She starts by creating the secret (s, p), where s is the x-coordinate of q above and p is just the point p already returned by the secret function. She returns p and $f\ s\ m$, where f is the encryption function, s, the secret, and m, the message to be encrypted.

Now, what does Bob do with this stuff to decrypt the message? Here it is:

```
eciesDecrypt :: ECIESParams → Natural → (Point, Natural) → Natural
eciesDecrypt (ECIES c _ _ f) k (p, cm) = let s = xco (mul c k p) in f s cm
```

For decryption, he needs the parameters, his own private key and the tuple (*Point*, *Natural*) generated by Alice. He does the following: he multiplies the private key (k) with the point p. When we go back, we see that this point p resulted from multiplying the primitive element g by a random number r. The public key, however, is also a product of a random number, namely, Bob's private key, and the generator. The secret, s was generated by multiplying Bob's public key by r. When Bob multiplies the point p with his private key k, he hence derives the same point.

To make that a bit clearer, let us adopt a better terminology. We will write points with capital letters and natural numbers with small letters. We then have G, the primitive element, K, Bob's public key, P, a point generated by Alice and Q, another point generated by Alice. We also have k, Bob's private key and r, a random number generated by Alice.

Alice computes: $P = rG$ and $Q = rK$. The x-coordinate of the latter is the shared secret. K, Bob's public key is kG. Q, hence, is $Q = rkG$. Bob, when decrypting computes kP, where $P = rG$. He, hence, computes krG and, since our group is associative and commutative, that is just Q whose x-coordinate is the shared secret.

Here comes a simple testing function that puts all the bits together. The function uses the previously defined curve $c1$ with generator $p1$:

```
eciesTest :: Bool → Natural → IO Bool
eciesTest verbose m = do
  (pri, ps@(ECIES _ _ pub _)) ← eciesMakeParams c1 p1 xor
  when verbose (do
    putStrLn $ "private: " ⧺ show pri
    putStrLn $ "public : " ⧺ show pub)
  (p, cipher) ← eciesEncrypt ps m
  when verbose (do
    putStrLn $ "cipher : " ⧺ show cipher
    putStrLn $ "p      : " ⧺ show p)
  return (eciesDecrypt ps pri (p, cipher) ≡ m)
```

11.7. EC Digital Signature Algorithm

The EC Digital Signature Algorithm, ECDSA, is less complex than ECIES in terms of parameters. But it is mathematically much more interesting. The objective of this algorithm is to provide message authentication, *i.e.* a scheme to sign and verify messages. As ECIES, it uses a pair of a private and a public key. The private key is used for signing and the public key is used to verify the signature, just as in RSA.

We start to describe the math of signing and verification. We have the following components (besides the usual parameters curve c and generator G): a random number r called the ephemeral key, a point $P = rG$ and its x-coordinate a. The private key k, a number, the public key K, a point computed as kG and the message m. The signature consists of the pair (a, s). We compute s as

$$s \equiv (m + ak)r' \pmod{o}, \tag{11.23}$$

where r' is the inverse of r in the group established by o the order of the group G, which itself must be a prime number.

The challenge for verification is to compute $P = rG$ and to compare the result with the x-coordinate a of P without having access to the private key k. To achieve this, we transform the equation above to get rid of k. We start by multiplying by r on both sides of the equation:

$$rs \equiv (m + ak) \pmod{o}. \tag{11.24}$$

We then multiply by the inverse of s:

$$r \equiv ms' + as'k \pmod{o}. \tag{11.25}$$

Now we multiply the generator G on both sides of this equation. Note that the values that appear in the equation are modulo o, which is the order of the group generated by G and, hence, the result is still in the group of the elliptic curve.

$$rG = ms'G + as'kG. \tag{11.26}$$

kG, the product of private key and generator, however, is just the public key. We can thus simplify to

$$rG = ms'G + as'K. \tag{11.27}$$

All the values on the right-hand sight of the last equation are known without knowing the private key. G is the generator, which is part of the parameters; m is the message to be verified; s' is the inverse of the second element of the signature, which can be computed using the order of the group generated by G. a, finally, is the first part of the signature. We, hence, can compute rG as $ms'G + as'K$. rG, however, is the point, P, from which the x-coordinate a was taken. If the x-coordinate of the result of our computation equals a, the signature is correct. Otherwise, it is a forgery.

Let us put the mathematics into code. We start, as usual, with the parameters:

```
data ECDSAParams = ECDSA Curve Point Point
```

The parameters consists of the curve, the generator and the public key. Next, we define a function to generate the key pair:

```
ecdsaKeyPair :: Curve → Point → IO (Natural, Point)
ecdsaKeyPair c g = do
  k ← randomNatural (2, o − 1)
  return (k, mul c k g)
  where o = gorder c g
```

This is nothing new. We generate a random number in the order of the group of the elliptic curve and multply the generator by this number. The number is the private key, k, and the resulting point is the public key, which we will call q in the following.

Here comes the function that creates the key pair and the parameters:

```
ecdsaMakeParams :: Curve → Point → IO (Natural, ECDSAParams)
ecdsaMakeParams c g = do
  (k, q) ← ecdsaKeyPair c g
  return (k, ECDSA c g q)
```

Now, we implement the sign function:

```
ecdsaSign :: ECDSAParams → Natural → Natural → IO (Natural, Natural)
ecdsaSign ps@(ECDSA c g q) k m = do
  r ← randomNatural (2, o − 1)
  if s1 ≡ 0 then ecdsaSign ps k m
            else return (a, s)
  where o  = gorder c q
        a  = xco (mul c r g)
        r' = inverse r o
        s1 = (m + k ∗ a) `mod` o
        s  = (s1 ∗ r')    `mod` o
```

We start by creating the so called ephemeral key, a random number r in the order of the group. We generate a point and get its x-coordinate a. We then get the inverse of r in the group o. Note that we treat the numbers in this range as the remainders of a prime number o. The order of the group, therefore, must be a prime number.

We now compute s in two steps. First, we compute $m + ak$, the message added to the product of the private key and the a we just computed modulo o. We then multiply the result by r', the inverse of r. Should $s1$ be a multiple of o, we try again with another r. Note that, if $s1$ is not a multiple of o, then $s1 \times r'$ is not a multiple of o either, since r' is from the remainder group of o. If o is a prime, this number and o do not share divisors and there is not way to get a multiple of o by multiplying by another number that does not share divisors with o. Otherwise, we return the pair (a, s). This is the signature.

Verification:

```
ecdsaVerify :: ECDSAParams → (Natural, Natural) → Natural → Bool
ecdsaVerify ps@(ECDSA c g q) (a, s) m = x ≡ a
  where o  = gorder c g
        s′ = inverse s o
        u  = (s′ ∗ m) ‘mod‘ o
        v  = (s′ ∗ a)  ‘mod‘ o
        x  = xco p
        p  = add c (mul c u g)
                   (mul c v q)
```

The function receives the paramters, the signature pair and the message. It computes the inverse of s and two variables u and v as $u = ms'$ and $v = as'$, both modulo o. Multiplying the generator by u results in the point $ms'G$; multiplying the public key q by v results in the point $as'K$. Their sum $ms'G + as'K$ equals rG, the point whose x-coordinate is a. Finally, we compare the result x with a. The comparison verifies the signature.

Here is a test function that brings all the bits together:

```
ecdsaTest :: Bool → Natural → IO Bool
ecdsaTest verbose m = do
  (k, ps) ← ecdsaMakeParams c1 p1
  when verbose (do
    putStrLn $ "private: " ++ show k
    putStrLn $ "public : " ++ show q)
  sig ← ecdsaSign ps k m
  when verbose (
    putStrLn $ "sig    : " ++ show sig)
  return (ecdsaVerify ps sig m)
```

11.8. Cryptoanalysis

11.9. Mr. Frobenius

11.10. Mr. Schoof

11.11. Mr. Elkies and Mr. Aktin

11.12. EC in Practice

12. Complex Numbers

12.1. i

12.2. Complex Numbers

12.3. The complex Plane

12.4. Polar Form

12.5. $\mathbb{C}$

12.6. Gaussian Integers

12.7. $\mathbb{Q}(i)$

12.8. ...

12.9. Complex Vector Spaces

12.10. ...

12.11. Quantum Mechanics

12.12. Hypercomplex Numbers

13. Quadratics, Cubics and Quartics

13.1. Quadratic Equations

13.2. The fundamental Theorem of Algebra

13.3. Cubics and Quartics

13.4. Quintics

13.5. Galois Theory

13.6. Field Theory

13.7. Postmodern Algebra

13.8. Category Theory

14. Euclid and beyond

14.1. Euclid’s Geometry

14.2. The Parallel Postulate

14.3. Non-Euclidian Geometries

14.4. Lost in Hilbert Space

14.5. Geometry and Relativity

14.6. Affine Geometry

14.7. Projective Geometry

14.8. Graph Theory

14.9. Topology

Part III.

Analysis

15. The Calculus

15.1. The Method of Exhaustion

15.2. Area below a Curve

15.3. Basic Rules of Integration

15.4. Minima, Maxima and Slopes

15.5. Basic Rules of Derivation

15.6. The fundamental Theorem of the Calculus

15.7. More Rules

16. Curve Sketching

17. Advanced Function Analysis

18. Generating Functions

19. Applied Mathematics

20. Prospects