Math for Programmers

Tobias Schoofs

April 9, 2014

1 Introduction

2 Brief Introduction to Haskell

3 Numbers

3.1 What are numbers and what should they be?

3.2 Peano Numbers

3.3 Decimal Numbers

3.4 Natural Numbers

3.5 Fun with natural numbers

3.6 Powers, Roots and Logarithms

4 Induction, Series 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* 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 it reaches a limit that will not influence the overall result anymore:

$$\begin{aligned}
&\mathit{converge} :: \mathit{Natural} \to (\mathit{Natural} \to \mathit{Natural}) \to \mathit{Natural} \to \mathit{Natural} \\
&\mathit{converge}\ l\ f\ n = \mathbf{let}\ r = f\ n \\
&\qquad\qquad \mathbf{in\ if}\ r \equiv l\ \mathbf{then}\ r \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ r + \mathit{converge}\ l\ f\ (n+1)
\end{aligned}$$

The function *converge* receives a *limit* l, which would be 0 in our case, a function f that transforms a natural number into another natural number and the natural number n that is actually applied to f. We compute the result r of $f\ n$ and if this result equals the limit, we produce the result r, otherwise, we continue with $n + 1$. This convergence function is very handy for our case and potentially for others we will encounter in the future, but we already stumble on concepts far ahead on our way like limits. Anyhow, let us look at how to use the convergence function:

$$
\begin{aligned}
&\mathit{ragmen} :: \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural} \\
&\mathit{ragmen}\ n\ m\ c = 1 + \mathit{converge}\ 0\ (f\ n\ m\ c)\ 1 \\
&\quad \textbf{where}\ f :: \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural} \to \mathit{Natural} \\
&\qquad\quad f\ n\ m\ c\ k = (n - m)\ `\mathit{floorDiv}`\ (c \uparrow k)
\end{aligned}
$$

The *ragmen* function consists in adding one to the result of *converge* with 0 as limit and a function f that receives four arguments, n, the number of customers, m, the number of endpoints, the ragman capacity c and the exponent k. The final 1 is the starting point for *converge*, *i.e.* the first k value. We pass f with n, m and c to *converge*, since *converge* expects a function of type $\mathit{Natural} \to \mathit{Natural}$. *converge* will then call f with the current value of k yielding one step result after the other.

4.2 Induction

The series we looked at in the previous section, for realistic values, converge very soon 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 divisable by 2. Even numbers could, hence, be expressed as $2k$ for all ks from $1 \ldots n$. Odd numbers, correspondingly, can be described as: $2k - 1$. The sum of the first n odd numbers is hence 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 $1 + 4 - 1$, hence 4, for $n = 3$, the formula

leads to $4 + 6 - 1 = 9$ and for $n = 4$, the result is $9 + 8 - 1 = 16$. All values correspond to the results we obtained above.

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

$$
\begin{aligned}
&\mathit{oddSum1} :: \mathit{Natural} \to \mathit{Natural} \\
&\mathit{oddSum1}\ n = \mathit{go}\ 1 \\
&\quad \textbf{where}\ \mathit{go}\ k \mid k > n \quad\ = 0 \\
&\qquad\qquad\quad \mid \mathit{otherwise} = (2 * k - 1) + \mathit{go}\ (k + 1)
\end{aligned}
$$

Now, is there a closed form that spares us from going to 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*.

A proof by induction proves that a property P holds for certain numbers by proving that it holds for a base case, $P(n)$, and by advancing from the base case to the next number, also holds for all other numbers we are interested in. Proofs by induction, therefore, 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 is $\mathbf{n^2}$, n times 1 is $\mathbf{n}$, 1 times n is $\mathbf{n}$ and 1 times 1 is $\mathbf{1}$. 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 1, *oddSum* is 1, *evenSum* is 2, *evenSum* is hence *oddSum* + 1 for this case; for 2, the difference between the results 4 and 6 is 2; for 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 one, $\sum_{k=1}^{1} 2k$ and $1 * (1 + 1)$ are both 2. Now assume that for 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 sum 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 is $\mathbf{n^2}$, n times 2 is $\mathbf{2n}$; 1 times n is $\mathbf{n}$, 1 times 2 is $\mathbf{2}$; 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 kind 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 this is not true. Note that the sum of the first n either odd or even numbers is in fact a greater number than the first n numbers, since, when we leave out every second number, than the result of counting n numbers is much higher than counting all numbers, *e.g.* for $n = 3$, the odd numbers are $1, 3, 5$, the even are $2, 4, 6$, all 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.*$\sum_{k=1}^{n} (k) = \frac{n(n+1)}{2}$. This 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 $\frac{2(n+1)}{2} = \frac{2n+2}{2}$. We can now add the two fractions:

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

After multiplying the numerator of the first one out $(n^2 + n)$ and then adding the two numerators we meet someone again we already know from the proof for even numbers:

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

The sums of the first n natural numbers in Haskell:

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

We will now turn our attention to somewhat more useful series.

4.3 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:

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

We start with `gcd 89 55`, which results in `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, computed the remainder and, since the remainder of two numbers with a distance among them that is less than the lower number is just the difference of the two numbers, we got to the next pair, 55 and 34, and 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 an 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, 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 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. The results are given in the third row. We should remove one from the final result 376, since this number includes the first pair that was already there.

If we define the Fibonacci function as

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

we can define a series $\sum_{k=2}^{12} F_k$, which, for the rabbit example, gives 375.

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 cases recursively as the sum of its two predecessors. In Haskell this looks like:

$$\begin{aligned} &\mathit{fib} :: \mathit{Natural} \to \mathit{Natural} \\ &\mathit{fib}\ 0 = 0 \\ &\mathit{fib}\ 1 = 1 \\ &\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \end{aligned}$$

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 – habit.

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 $F_{n+1} + F_{n+2}$:

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

[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.

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 this concludes the proof.

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, a certain very strange triangle.

4.4 Factorial

A fundamental concept in mathematics, computer science and real life is the idea of *permutation*, which refers to 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]`, 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: `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!

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

$$\begin{aligned}
&\mathit{insall} :: a \to [a] \to [[a]] \\
&\mathit{insall}\ p\ [\,] \quad\ = [[p]] \\
&\mathit{insall}\ p\ (z : zs) = (p : z : zs) : (\mathit{map}\ (z:)\ \$\ \mathit{insall}\ p\ zs)
\end{aligned}$$

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 `z:zs`, we add p in front of the list (`p:z:zs`) and then repeat the process for all possible reductions of the list until we reach the base case. In each recursion step, we add z, the head in front of the resulting lists. Imagine this for the case $p = 1$, $z = 2$ and $zs = \{3\}$: We first create `[1,2,3]` by means of `p:z:zs`; we then enter *insall* again with $p = 1$, $z = 3$ and $zs = \{\}$, which creates `1:3:[]`, to which later, when we return, 2, the z 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 $z = 3$, mapping `(z:)` 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 (between 2 and 3) and at its 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:

$$
\begin{aligned}
&perms :: [a] \to [[a]] \\
&perms\ [\,] = [[\,]] \\
&perms\ (x : xs) = concatMap\ (insall\ x)\ \$\ perms\ xs
\end{aligned}
$$

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 giving `[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*, however, 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 written:

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

We can define factorial in Haskell as follows:

$$\begin{aligned} &fac :: (Num\ a, Eq\ a) \Rightarrow a \rightarrow a \\ &fac\ 0 = 1 \\ &fac\ n = n * fac\ (n - 1) \end{aligned}$$

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. The result of any factorial would always 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 five factorials, which you can create by `map fac [1..5]`, are: $1, 1, 2, 6, 24, 120$. Then it increases 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 a very expensive operation.

In the literature, different notations are used to describe permutations. A very simple, but quite verbose notation is the *two-line* notation used by the great French mathematician Augustin-Louis Cauchy (1789 - 1857). In this notation, the original list is given in one line and the resulting list in a second line, hence, for a permutation σ:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 3 & 1 \end{pmatrix}. \tag{4.16}$$

According to this definition, the permutation σ would substitute 1 by 2, 2 by 5, 3 by 4 and 4 by 3 and 5 by 1. The alternative *tuple notation* would just give the second line as (2,5,4,3,1) and assume a *natural* ordering. 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 $(x\ \sigma(x)\ \sigma(\sigma(x))\ \sigma(\sigma(\sigma(x)))\ \ldots)$, leaving out the final step where the cycle returns to the beginning. 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 at its original place. $\sigma(3) = 4$ and $\sigma(4) = 3$. This describes the permutation σ completely.

Can we device 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. Whenever p is found in the original, s is placed at this postition in the resulting list and the new resulting list is returned. 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:

$$\textbf{type}\ Perm\ a = [[a]]$$

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:

$$\begin{aligned}
&permute :: (Eq\ a) \Rightarrow Perm\ a \rightarrow [a] \rightarrow [a]\\
&permute\ [\,]\ \ \ \ \ \ \ \ xs = xs\\
&permute\ ([\,] : ps)\ \ xs = permute\ ps\ xs\\
&permute\ ([p] : ps)\ xs = permute\ ps\ xs\\
&permute\ (p : ps)\ \ \ xs = permute\ ps\ \$\ orbit\ (head\ p)\ xs\ p\\
&\quad \textbf{where}\ orbit\ _\ rs\ [\,]\ \ \ \ \ \ \ \ \ \ \ \ = rs\\
&\qquad\qquad orbit\ x\ rs\ [u]\ \ \ \ \ \ \ \ \ \ \ = replace\ u\ x\ xs\ rs\\
&\qquad\qquad orbit\ x\ rs\ (p1 : p2 : pp) = orbit\ x\ (replace\ p1\ p2\ xs\ rs)\ (p2 : pp)
\end{aligned}$$

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 *orbit* 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 c = [[1,2,5][3,4]]
permute c [1,2,3,4,5] is [2,5,4,3,1].
permute c [2,5,4,3,1] is [5,1,3,4,2].
```

```
permute c [5,1,3,4,2] is [1,2,4,3,5].
permute c [1,2,4,3,5] is [2,5,3,4,1].
permute c [2,5,3,4,1] is [5,1,4,3,2].
permute c [5,1,4,3,2] is [1,2,3,4,5].
```

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 $2*3 = 6$ times.

One could think of systems of permutations (and people have actually done so), such that the application of these permutations to each other would always yield the same set of lists. 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

$$e = (1)(2)(3)(4) \tag{4.17}$$
$$a = (1\ 2)(3)(4) \tag{4.18}$$
$$b = (1)(2)(3\ 4) \tag{4.19}$$
$$c = (1\ 2)(3\ 4) \tag{4.20}$$

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 are called *groups*. A group consists of a set of objects (numbers, permutations, *etc.*) and a binary operation with the property that the result of the application on members of the set is a member of the set again. We say that the group is closed under this operation, *e.g.* permutation groups are closed under composition. The group operation must fulfil associativity, that is: $x \cdot (y \cdot z) = (x \cdot y) \cdot z = x \cdot y \cdot z$, which is clearly fulfilled for the permutations above. For natural numbers and addition or multiplication, this is true as well.

Additionally, a group must contain an identity for this operation, that is an element of the set that, combined with another element of the set using the operation, yields this element, or: $e \cdot x = x \cdot e = x$. For natural numbers and addition, the identiy is 0, for multiplication, it is 1. Every element in the set, must also contain an inverse element, such that $x \cdot y = y \cdot x = e$. This is true for the permutations above, since each permutation is the inverse of itself: $a \cdot a = e$. This, however, is not true for natural numbers. Therefore, natural numbers do not form a group! They just form a *monoid*.

The set of all possible permutations of a list is also a group, which is called the *symmetry group*: it has an identiy element (the permutation that fixes all elements), it has for every element an inverse element (since all possible permutations are in the group, there is for each permutation a permutation that returns to the original configuration), composition is associative and, 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. But, of course, not all possible subsets of the symmetry group are groups. Subsets of the symmetry 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, as well, are not groups.

The group concept that was developed during the 18^{th} and 19^{th} century by Joseph-Louis Lagrange, Paolo Ruffini, Niels Henrik Abel and Évariste Galois, has led to results that are fundamental in many areas of mathematics, in particular algebra. We will meet the group concept again, but for the moment, we will have to say goodbye to turn our attention to a tremendously important subject.

4.5 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, finally, remove 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 permutation 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 come to $\{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$, *i.e.* three different selections of two elements out of three.

This algorithm suggests that the number of k selections out of n elements is 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 – $\{1, 2\}$ and $\{2, 1\}$ represent the same selection. 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 further 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 adequate 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 conditioned by the first choice. For the case 2 out of 3, the first selection halves our options in the second step. The correct formula for

selections is therefore not, as with permutations, $n \times (n-1)$, but $\frac{n}{1} \times \frac{n-1}{2}$, which, for the case 2 out of 3, is $\frac{3}{1} \times \frac{2}{2} = 3 \times 1 = 3$. When we continue this scheme, considering that each choice that was already made conditions the next choice, we get a product of the form: $\frac{n}{1} \times \frac{n-1}{2} \times \frac{n-2}{3} \times \ldots$ 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: $\frac{n}{1} \times \frac{n-1}{2} \times \cdots \times \frac{n-(k-1)}{k}$. 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, called falling factorial or *to-the-k^{th}-falling*:

$$n^{\underline{k}} = n \times (n-1) \times \cdots \times (n-k+1) = \prod_{j=1}^{k} n+1-j. \tag{4.21}$$

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

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j} \tag{4.22}$$

or:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{4.23}$$

But there is also the following formula, which is closer to our first intuition that the selection is somehow related to factorials reduced by some value:

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!}. \tag{4.24}$$

The first two equations, even if mathematically equivalent with this one, are obviously more attractive, when it comes to efficiently implementing the binomial coefficient. We will immediately return to this questions, but let us first look at some concrete values of the binomial coefficients: $\binom{n}{0} = \binom{n}{n} = 1$ and $\binom{n}{1} = \binom{n}{n-1} = n$. 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 who used binomial coefficients to investigate probabilities and, in the process, created a new branch in mathematics, probabilistics:

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 n in $\binom{n}{k}$. The left-most value in each line, represents the value $\binom{n}{0} = 1$, and right-most value is $\binom{n}{n} = 1$. The values between the outermost ones, represent the values for $\binom{n}{1} \ldots \binom{n}{n-1}$. 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 symmetrical, *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.*

$$\binom{n}{k} = \begin{cases} \binom{n}{n-k} & \text{if } 2k > n \\ \prod_{j=0}^{k} \frac{n+1-j}{j} & \text{otherwise} \end{cases} \tag{4.25}$$

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:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}, \tag{4.26}$$

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

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*. In-

deed, 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 method for multiplication we have learnt in chapter 1 can be rephrased as

$$(a+b)(c+d) = ac + ad + bc + bd \tag{4.27}$$

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

$$(a+b)(a+b) = aa + ab + ba + bb, \tag{4.28}$$

which is the same as

$$(a+b)(a+b) = a^2 + 2ab + b^2. \tag{4.29}$$

When we now multiply $(a+b)$ again, we get:

$$\begin{aligned}(a+b)(a^2+2ab+b^2) = a^3 + 2a^2b + ab^2 + ba^2 + 2ab^2 + b^3 = \\ a^3 + 2a^2b + ba^2 + ab^2 + 2ab^2 + b^3 = \\ a^3 + 3a^2b + 3ab^2 + b^3\end{aligned} \tag{4.30}$$

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

$$\begin{aligned}(a+b)(a^3+3a^2b+3ab^2+b^3) = \\ a^4 + 3a^3b + 3a^2b^2 + ab^3 + ba^3 + 3a^2b^2 + 3ab^3 + b^4 = \\ a^4 + 3a^3b + ba^3 + 3a^2b^2 + 3a^2b^2 + ab^3 + 3ab^3 + b^4 = \\ a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4\end{aligned} \tag{4.31}$$

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:

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

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

$$\dots$$

This, in general, is the binomial theorem:

$$\begin{aligned}(x+y)^n &= \binom{n}{0}x^n y^0 + \binom{n}{1}x^{n-1}y^1 + \dots + \binom{n}{n}x^0 y^n \\ &= \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k}\end{aligned} \tag{4.32}$$

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 the 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 of three $\{1, 2, 3\}$: $(a + b)(aa + ab + ba + bb) = aaa + \mathbf{aa}b + a\mathbf{a}b + abb + b\mathbf{aa} + bab + bba + bb$.

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

$$(a+b)^{n+1} = \sum_{k=0}^{n+1}\binom{n+1}{k}a^k b^{n+1-k}. \tag{4.33}$$

We start with the simple equation

$$(a+b)^{n+1} = (a+b)^n(a+b) \tag{4.34}$$

and then reformulate it introducing the base case:

$$(a+b)^{n+1} = \left(\sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}\right)(a+b). \tag{4.35}$$

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:

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

The second term looks already quite similar to the case in equation 4.33, both have $a^k b^{n+1-k}$. 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 rules of which we are sure that they hold. 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 $n+1$:

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

Please confirm for yourself with pencil and paper that the first summation in equations 4.36 and 4.37 remains 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.

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 a 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 $b^{n+1-(n+1)} = b^{n+1-n-1} = b^0 = 1$. The case $k = 0$ in the second term corresponds to $\binom{n}{0} a^0 b^{n+1-0} = b^{n+1}$. When we combine this, we get to:

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

The next step provides you with a test of how well you have internalised the distributive law. The two summations above have the following 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 $a^k b^{n+1-k}$ for $k = 1$ and d is 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 the 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 just apply these little steps on the summations:

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

Now you might recognise Pascal's rule given in equation 4.26 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 above equation to

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

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

$$(a+b)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} a^k b^{n+1-k} \tag{4.41}$$

and we are done, since – using some mathematical trickery – we have just derived equation 4.33.

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 4.22 and 4.23, which are repeated here for convenience:

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j}, \tag{4.42}$$

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{4.43}$$

The first option performs k divisions and $k - 1$ multiplications: one division per step and 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 multiplications for $n^{\underline{k}}$, $k - 1$ multiplications for $k!$ and one division, hence, $k + k - 1 + 1 = 2k$ operations. That is one step more but without the "minor" costs of the additional sum. 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 and without the need to store huge values in memory. So, let us implement the first option:

```
choose :: Natural → Natural → Natural
choose n 0 = 1
choose n 1 = n
choose n k | 2 * k > n  = choose n (n - k)
           | otherwise = go 1 1
  where go m i | i > k       = m
               | otherwise = go (m * (n - k + i)) `div` i) (i + 1)
```

The implementation is straight forward. The function *choose* is defined over natural

numbers (so we have not 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 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) `div` 1) (1+1),
```

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

```
go (3 * (5 - 3 + 2) `div` 2) (2+1),
```

which equals `go 6 3`, expands to

```
go (6 * (5 - 3 + 3) `div` 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 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 $\frac{6+1-1=6}{1} \times \frac{6+1-2=5}{2} \times \ldots$. The second factor, $\frac{5}{2}$, is not a natural number. We cannot express this value with natural numbers, 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.

4.6 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. Today many, if not even 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, 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, *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 kinds 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 in Haskell is list comprehension, for instance: `[x | x <- [1..100], even x, x ‘mod‘ 3 /= 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* $\{\}$, often denoted as $\emptyset$, that does not contain any elements.

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: $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

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 n, *i.e.* for one row of Pascal's Triangle. For $n = 0$, we have: $\binom{0}{0} = 1$, since the only subset of $\emptyset$ is $\emptyset$. 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 $a^n + \binom{n}{1}a^{n-1}b + \cdots + \binom{n}{n-1}ab^{n-1} + b^n$, 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:

$$
\begin{aligned}
&ps :: (Eq\ a) \Rightarrow [a] \to [[a]] \\
&ps\ [\,] = [[\,]] \\
&ps\ (x : xs) = ps\ xs +\!\!+ map\ (x:)\ \$\ ps\ xs
\end{aligned}
$$

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 imediately 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: `[[], [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 uses binary numbers to represent powersets. This 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. Binary numbers, which we will discuss in more detail later, use only the digits 0 and 1 instead of the digits $0 \ldots 9$ 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 persons 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 as lists – like that of 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: $1024 = 1 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + \cdots + 0 \times 2^0$ 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:

$$
\begin{aligned}
&\mathit{natLog} :: \mathit{Natural} \to \mathit{Natural} \to (\mathit{Natural}, \mathit{Natural}) \\
&\mathit{natLog}\ b\ n = \mathbf{let}\ e = \mathit{floor}\ \$\ \mathit{logBase}\ (\mathit{fromIntegral}\ b) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{fromIntegral}\ n) \\
&\qquad\qquad\quad \mathbf{in}\ (e, n - 2 \uparrow e)
\end{aligned}
$$

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:

$$
\begin{aligned}
&\mathit{binExp} :: \mathit{Natural} \to [\mathit{Natural}] \\
&\mathit{binExp}\ 0 = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{binExp}\ 1 = [0] \\
&\mathit{binExp}\ n = \textbf{let}\ (e, r) = \mathit{natLog}\ 2\ n\ \textbf{in}\ e : \mathit{binExp}\ r
\end{aligned}
$$

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`:

`binExp 1000` $= (9, 1000 - 2^9 = 1000 - 512 = 488)$
`9 :` `binExp 488` $= (8, 488 - 2^8 = 488 - 256 = 232)$
`9 :` `8 :` `binExp 232` $= (7, 232 - 2^7 = 232 - 128 = 104)$
`9 :` `8 :` `7 :` `binExp 104` $= (6, 104 - 2^6 = 104 - 64 = 40)$
`9 :` `8 :` `7 :` `6 :` `binExp 40` $= (5, 40 - 2^5 = 40 - 32 = 8)$
`9 :` `8 :` `7 :` `6 :` `5 :` `binExp 8` $= (3, 8 - 2^3 = 8 - 8 = 0)$
`9 :` `8 :` `7 :` `6 :` `5 :` `3 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 $0 \ldots 2^n$, 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:

$$
\begin{aligned}
&\mathit{ps2} :: (\mathit{Eq}\ a) \Rightarrow [a] \rightarrow [[a]] \\
&\mathit{ps2}\ [\,] = [[\,]] \\
&\mathit{ps2}\ \mathit{xs} = \mathit{go}\ (2 \uparrow (\mathit{length}\ \mathit{xs}) - 1)\ 0 \\
&\quad \textbf{where}\ \mathit{go}\ n\ i \mid i \equiv n \quad\quad = [\mathit{xs}] \\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \textbf{let}\ s = \mathit{map}\ \mathit{exp2idx}\ \$\ \mathit{binExp}\ i \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ s : \mathit{go}\ n\ (i + 1) \\
&\qquad\quad \mathit{exp2idx}\ x = \mathit{xs}\ !!\ (\mathit{fromIntegral}\ x)
\end{aligned}
$$

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: $2^{(\mathit{length}\ \mathit{xs})} - 1$, which is the greatest number that can be represented with a binary number with n digits, when we start to count from 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 a mapping function *exponent* $\rightarrow$ *index* to the result of *binExp* applied to the current number i. The mapping function, *exp*2*idx*, 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 arrays, and, of course, a more efficient number representation than our humble natural

numbers, the algorithm definitely beats the one we implemented as *ps*. Unfortunately, lists show very bad performance with random access such as indexing. Therefore, *ps*2 is slower than *ps*. But using Haskell vectors (implemented in module *Data.Vector*) and Integers instead of our *Natural*, *ps*2 is indeed faster. The changes, by the way, are minimal. Just compare the implementation of *ps*2 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 $ binExp i
                               in s : go v n (i + 1)
        exp2idx x = v ! (fromIntegral x)
```

The changes to the code of *ps*2 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 huge 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 different $A \setminus B$ 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 list will introduce duplicates for any $x \in a \wedge x \in b$.

Intersect is slightly more difficult. The intersect of two sets a and b is the set that contains all elements that are in both sets, a and b. We could implement this by means of *nub* as well, since 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 \\ 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 \\ nub (A ++ B) =
[1,2,3] ++ [3,4,5] \\ nub ([1,2,3] ++ [3,4,5]) =
[1,2,3,3,4,5] \\ nub ([1,2,3,3,4,5]) =
[1,2,3,3,4,5] \\ [1,2,3,4,5]) =
[3].
```

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

$$
\begin{aligned}
&\mathit{intersect} :: (\mathit{Eq}\ a) \Rightarrow [a] \to [a] \to [a] \\
&\mathit{intersect}\ [\,]\ \_ = [\,] \\
&\mathit{intersect}\ \_\ [\,] = [\,] \\
&\mathit{intersect}\ (a : as)\ bs \mid a \in bs \quad\ = a : \mathit{intersect}\ as\ bs \\
&\qquad\qquad\qquad\qquad\ \mid \mathit{otherwise} = \quad \mathit{intersect}\ as\ bs
\end{aligned}
$$

We first define the intersection of the empty set with any other set as the empty set. (Note the similarity of the $\emptyset$ in union and intersection with 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, whose remainder results from the application of *intersect* on the tail of the first list; otherwise, we only construct the remainder of the result set 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 $S = \{s_1, s_2, \ldots, s_n\}$ of subsets of U, $s_1 \subseteq U, s_2 \subseteq U, \ldots, s_n \subseteq U$, such that the union of all the sets in S equals U, $s_1 \cup s_2 \cup \cdots \cup s_n = U$. 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 runs. No real-world heuristics are available for the kernel to find an optimal balance.

There are, however, also purely mathematical heuristics. 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 solution, 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 length of the difference between the universe and another set:

$$dist :: (Eq\ a) \Rightarrow [a] \rightarrow [a] \rightarrow Int$$
$$dist\ a\ b = length\ (a \setminus\setminus b)$$

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 r 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 $\emptyset$ and the universe.

The second parameter passed to *loop*, rs, is the union of all 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 $\emptyset$. 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 $m' < m$, then 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 $n + n - 1 + n - 2 + \cdots + 1$ or simply the series $\sum_{k=1}^{n} k$, which, as we already know, 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.

5 Primes

6 Fractions

7 Negative Numbers

7.1 Negative Numbers

7.2 Boolean Algebra

7.3 Integers

7.4 Fractions

7.5 Groups, Rings and Fields

8 Number Systems

8.1 Binary Numbers

8.2 Octal Numbers

8.3 Hexadecimal Numbers

8.4 Any Number's System

8.5 Polynomials

9 Polynomials

10 Functions and the Cartesian Plane

11 Linear Equations

12 Quadratics, Cubics and Quartics

12.1 Quadratic Equations

12.2 Real Numbers

12.3 Complex Numbers

12.4 Generating Functions

12.5 Cubic Equations

12.6 Quartic Equations

12.7 Revisiting Fields

13 Galois

14 Calculus

14.1 Rates of Change

14.2 Derivatives

14.3 Integrals