1 Induction, Series, Sets and Combinatorics

1.1 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:

$$r = \left\lceil \frac{n-m}{c} \right\rceil + \left\lceil \frac{n-m}{c^2} \right\rceil + \ldots \tag{1.1}$$

Or even nicer:

$$r = \sum_{k=1}^{\infty} \left\lceil \frac{n-m}{c^k} \right\rceil \tag{1.2}$$

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 1.2, 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:

$$r = 1 + \sum_{k=1}^{\infty} \left\lfloor \frac{n-m}{c^k} \right\rfloor \tag{1.3}$$

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 $\left\lfloor \frac{1500}{10} \right\rfloor + \left\lfloor \frac{1500}{100} \right\rfloor + \left\lfloor \frac{1500}{1000} \right\rfloor + 1 = 167$ ragmen and will have 333 endpoints *on*. For 5.000 customers, the series, again, converges for $k = 4$ and we will need $\left\lfloor \frac{4500}{10} \right\rfloor + \left\lfloor \frac{4500}{100} \right\rfloor + \left\lfloor \frac{4500}{1000} \right\rfloor + 1 = 500$, which is just the amount of endpoints we have available in total. We, thus, cannot serve 5.000 customers with this configuration. We would need to increase c and accept more workload in moving goods into wating zones.

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

$$\begin{aligned}
&\mathit{converge1} :: (\mathit{Natural} \to \mathit{Natural}) \to \mathit{Natural} \to \mathit{Natural} \\
&\mathit{converge1}\ f\ n = \mathbf{let}\ r = f\ n \\
&\qquad\qquad \mathbf{in\ if}\ r \equiv 0\ \mathbf{then}\ r \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ r + \mathit{converge1}\ l\ f\ (n+1)
\end{aligned}$$

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

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

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

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

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

$$
\begin{aligned}
&\mathit{convSum} :: (\mathit{Natural} \to \mathit{Natural}) \to \mathit{Natural} \to \mathit{Natural} \\
&\mathit{convSum} = \mathit{converge}\ 0\ (+) \\
&\mathit{convProduct} :: (\mathit{Natural} \to \mathit{Natural}) \to \mathit{Natural} \to \mathit{Natural} \\
&\mathit{convProduct} = \mathit{converge}\ 1\ (*)
\end{aligned}
$$

Let us look at how to use the convergence function:

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

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

1.2 Induction

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

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

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

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

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

```
oddSum1 :: Natural → Natural
oddSum1 n = go 1
  where go k | k > n      = 0
             | otherwise = (2 * k - 1) + go (k + 1)
```

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

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

we see that all the numbers are perfect squares: $1, 4, 9, 16, 25, 36, 49, 65, 81$. Indeed, the results suggest that the sum of the first n odd numbers equals n^2. But is this always

true or does it hold only for the first nine numbers we just happened to look at? Let us try a proof by *induction*.

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

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

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

$$\sum_{k=1}^{n+1}(2k-1) = (n+1)^2. \tag{1.4}$$

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:

$$\sum_{k=1}^{n}(2k-1) + 2(n+1) - 1 = (n+1)^2. \tag{1.5}$$

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:

$$n^2 + 2n + 1 = (n+1)^2 \quad \square \tag{1.6}$$

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

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

$$
\begin{aligned}
&\mathit{oddSum} :: \mathit{Natural} \to \mathit{Natural}\\
&\mathit{oddSum} = (\uparrow 2)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{evenSum1} :: \mathit{Natural} \to \mathit{Natural}\\
&\mathit{evenSum1}\ n = \mathit{go}\ 1\\
&\quad \textbf{where}\ \mathit{go}\ k \mid k > n \quad\ = 0\\
&\qquad\qquad\ \ \mid \mathit{otherwise} = 2 * k + \mathit{go}\ (k+1)
\end{aligned}
$$

Applying $\mathit{evenSum1}$ to the test set [1..9] gives the sequence: $2, 6, 12, 20, 30, 42, 56, 72, 90$. These are obviously no perfect squares and, compared to the odd numbers $(1, 4, 9, 16, \dots)$, the results are slightly greater. How much greater are they? For $n = 1$, oddSum is 1, $\mathit{evenSum}$ is 2, $\mathit{evenSum}$ is hence $\mathit{oddSum} + 1$ for this case; for $n = 2$, the difference between the results 4 and 6 is 2; for $n = 3$, the difference between 9 and 12 is 3. This suggests a pattern: the difference between oddSum and $\mathit{evenSum}$ is exactly n. This would suggest the closed form $n^2 + n$ or, which is the same, $n(n+1)$. Can we prove this by induction?

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

$$\sum_{k=1}^{n+1} 2k = (n+1)(n+2). \tag{1.7}$$

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

$$\sum_{k=1}^{n} (2k) + 2(n+1) = (n+1)(n+2). \tag{1.8}$$

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

$$n(n+1) + 2(n+1) = (n+1)(n+2). \tag{1.9}$$

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:

$$n^2 + 3n + 2 = (n+1)(n+2) \quad \square \tag{1.10}$$

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

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

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

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

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

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

$$\sum_{k=1}^{n+1} k = \frac{(n+1)(n+2)}{2}. \tag{1.11}$$

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

$$\frac{n(n+1)}{2} + \frac{2n+2}{2} = \frac{(n+1)(n+2)}{2} \tag{1.12}$$

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

$$\frac{n^2 + 3n + 2}{2} = \frac{(n+1)(n+2)}{2} \quad \square \tag{1.13}$$

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

$$natSum :: Natural \to Natural$$
$$natSum = (`div`2) \circ evenSum$$

1.3 Arithmetic and Geometric Series

1.4 The Fibonacci Sequence

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

$$gcd :: Natural \to Natural \to Natural$$
$$gcd\ a\ 0 = a$$
$$gcd\ a\ b = gcd\ b\ (a\ `rem`\ b)$$

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

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

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

month, the first pair produces one more offsping, hence we have 3 pairs. In the third month, we have 2 fertile pairs producing each 1 more pair and we, hence, have 5 pairs. This, quickly, becomes confusing. Here a table that gives an overview of what happens during the first year:

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

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

The Fibonacci function can be defined as:

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

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

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

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 – property.

The first known practical application of the Fibonacci sequence in Europe appeared in an article of the French mathematician Gabriel Lamé in 1844 that identified the worst case of the Euclidian gcd algorithm as any two subsequent numbers of the Fibonacci

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

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

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

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

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

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

which, of course, simplifies to

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

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

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

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

1.5 Factorial

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

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

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

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

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

As base case, we have p, the new element, added to the empty list, which trivially results in $[[p]]$. From here on, for any list of the form $x : xs$, we add p in front of the list $(p : x : xs)$ and then repeat the process for all possible reductions of the list until we reach the base case. In each recursion step, we add x, the head of the original list, in front of the resulting lists. Imagine this for the case $p = 1$, $x = 2$ and $xs = \{3\}$: We first create $[1, 2, 3]$ by means of $p : x : xs$; we then enter *insall* again with $p = 1$, $x = 3$ and $xs = \{\}$, which creates $1 : 3 : [\,]$, to which later, when we return, 2, the x of the previous step, is

inserted, yielding $[2,1,3]$. With the next step, we hit our base case $insall\ 1\ [\,] = [[1]]$. Returning to the step with $x = 3$, mapping $(x{:})$ gives $[3,1]$ and, one step further back, $[2,3,1]$. We, hence, have created three cases: $[[1,2,3],[2,1,3],[2,3,1]]$ inserting 1 in front of the list, in the middle of the list and at the end.

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

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

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 yielding $[3]$; one step further back, *insall* 2 would now result in $[[2,3],[3,2]]$. Mapping *insall* 1 finally on these two lists leads to the desired result.

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

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

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

$$n! = \prod_{k=1}^{n} k. \tag{1.14}$$

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. Otherwise, the result of any factorial would be zero! The inversion of factorial, *i.e.*

$$n! = \frac{(n+1)!}{n+1}, \tag{1.15}$$

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

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

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

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

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

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

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

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

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

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

We will use *replace* in the definition of a function creating permutations according to a

definition in cycle notation. Cycle notation is translated to Haskell as a list of lists, each inner list representing one orbit:

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

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:

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

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 processes the orbit by replacing the first element by the second, the second by the third and so on. The last element is replaced by the head of the orbit, which, for this purpose, is explicitly passed to the function.

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

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

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

The result in the third line is funny: It is almost identical to the original list, but with 3 and 4 swapped. The two orbits of the permutation σ appear to move at different

speed: the first orbit with three elements needs three applications to return to the original configuration; the second orbit with two elements needs only two applications. Apparently, 2 does not divide 3; the orbits are therefore out of sink until the permutation was performed $2 \times 3 = 6$ times.

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

$$e = (1)(2)(3)(4) \tag{1.16}$$
$$a = (1\ 2)(3)(4) \tag{1.17}$$
$$b = (1)(2)(3\ 4) \tag{1.18}$$
$$c = (1\ 2)(3\ 4) \tag{1.19}$$

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

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

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

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

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

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

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

1.6 Random Permutations

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

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

When you think of it, it is indeed difficult to create a sequence without any *patterns* in

it, such as regular distances between elements, periodic repetitions and so on. You may think of any of the sequences we have looked at so far: there was always a pattern that led to a way to define a program to generate that sequence and the program was always represented as a finite string of Haskell code that was much shorter than the sequence, which, usually, was infinite. For instance, the definitions of the Fibonacci sequence or of Factorials are much shorter than that sequences, which are infinite. But, even with finite sequences, we have the same principle. Look, for instance, at the sequence $5, 16, 8, 4, 2, 1$, which, on the first sight, appears completely random. However, there is a program that generates this as well as many other similar sequences, namely the *hailstone* algorithm:

$$
\begin{aligned}
&\mathit{hailstone} :: \mathit{Integer} \to [\mathit{Integer}] \\
&\mathit{hailstone}\ 1 = [1] \\
&\mathit{hailstone}\ n \mid \mathit{even}\ n \quad\ = n : \mathit{hailstone}\ (n\ \text{`}\mathit{div}\text{`}\ 2) \\
&\qquad\qquad \mid \mathit{otherwise} = n : \mathit{hailstone}\ (3 * n + 1)
\end{aligned}
$$

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

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

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

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

Random number generators implemented in modern operating systems collect numbers that are created by the system while operating. A typical source of randomness is

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

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

$$
\begin{aligned}
&\mathit{randomNatural} :: (\mathit{Natural}, \mathit{Natural}) \to \mathit{IO\ Natural} \\
&\mathit{randomNatural}\ (l, u) = \mathbf{let}\ \mathit{il} = \mathit{fromIntegral}\ l \\
&\qquad\qquad\qquad\qquad\quad \mathit{iu} = \mathit{fromIntegral}\ u \\
&\qquad\qquad\qquad\quad\ \mathbf{in}\ \ \mathit{fromIntegral} <\$> \\
&\qquad\qquad\qquad\qquad\quad (\mathit{randomRIO}\ (\mathit{il}, \mathit{iu}) :: (\mathit{IO\ Integer}))
\end{aligned}
$$

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

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

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

in *Data.Vector.Mutable*. First, we will look at a function that creates a mutable vector from a list:

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

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

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

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

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

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

We start by creating the vector using the function *createVector* defined above. Note

that, since we need it more than once, we initially store the size of the list in the variable n. Since we compute it again in *createVector*, there is potential for improvement.

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

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

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

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

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

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

resulting overall in a valid permutation of the input list.

1.7 Binomial Coefficients

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

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

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

But note that, when we select 2 in the first row, the option 1 in the second row will vanish, since we already selected $\{1, 2\}$, which is the same as $\{2, 1\}$. Likewise, when we select 3 in the first row, we cannot select 1 in the third row because, again, $\{1, 3\}$ is the

same as $\{3, 1\}$. It, therefore, would be much more appropriate to represent our options as in the following table:

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

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

$$n^{\underline{k}} = n \times (n-1) \times \cdots \times (n-k+1) = \prod_{j=1}^{k} n + 1 - j. \tag{1.20}$$

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

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j} \tag{1.21}$$

or:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{1.22}$$

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

$$\binom{n}{k} = \frac{n!}{k! \times (n-k)!}. \tag{1.23}$$

It can be seen immediately that this formula is equivalent to equation 1.22, whenever $k \leq n$, since the values of $n!$ in the numerator cancel out with the values of $(n-k)!$ in the denominator. Indeed, $n!$ could be split into two halves (which are not necessarily equal of course), the upper product $n^{\underline{k}}$ $(n \times (n-1) \times \cdots \times (n-k+1))$ and the lower product $(1 \times 2 \times \cdots \times (n-k))$. By cancelling out the lower half, we remove the lower product from numerator and denominator and are left with the falling factorial in the numerator.

We could have derived equation 1.22 much more easily with a different kind of reasoning: Given a set with n elements, there are $n^{\underline{k}}$ permutations of k elements of this set. There are n ways to choose the first element, $n-1$ ways to choose the second element and so on and $n-k+1$ ways to choose the k^{th} element. Obviously, we could reach the same result, all permutations of k elements out of n, by first selecting these k elements and then create all possible permutations of these k elements. The number of possibilities of choosing k out of n is the binomial coefficient, $\binom{n}{k}$, which we would like to derive. The possible permutations of these k elements is of course $k!$ We now have to combine these two steps: We have for any selection of k elements out of n $k!$ permutations, that is $\binom{n}{k} \times k!$ Since this processing has the same result as choosing all permutations of k out of n in the first place, we come up with the equation:

$$n^{\underline{k}} = \binom{n}{k} \times k! \tag{1.24}$$

To know what the expression $\binom{n}{k}$ is we just divide $k!$ on both sides of the equation and get equation 1.22:

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{1.25}$$

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

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

In this triangle, each row represents the coefficients for one specific value of n in $\binom{n}{k}$. The left-most value in each line represents the value $\binom{n}{0} = 1$ and the right-most value is $\binom{n}{n} = 1$. The values between the outermost ones represent the values for $\binom{n}{1} \ldots \binom{n}{n-1}$. 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 symmetric, *i.e.* $\binom{3}{1} = \binom{3}{2} = 3$, $\binom{6}{2} = \binom{6}{4} = 15$, $\binom{7}{2} = \binom{7}{5} = 21$ or, in general, $\binom{n}{k} = \binom{n}{n-k}$. This is a strong hint how we can optimise the computation of the binomal coefficients. Indeed, whenever k in $\binom{n}{k}$ is more than the half of n, we can use the corresponding value from the first half of k's, *i.e.*

$$\binom{n}{k} = \begin{cases} \binom{n}{n-k} & \text{if } 2k > n \\ \prod_{j=0}^{k} \frac{n+1-j}{j} & \text{otherwise} \end{cases} \tag{1.26}$$

Thank you, Triangle!

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

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}. \tag{1.27}$$

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

A real light bulb moment, however, comes when realising the relation of binomial coefficients to multiplication. We discussed several times already that there are certain patterns in multiplication, which now turn out of have a name: *binomial coefficient*. 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 distributive law tells us that

$$(a+b)(c+d) = ac + ad + bc + bd. \tag{1.28}$$

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

$$(a+b)(a+b) = aa + ab + ba + bb, \tag{1.29}$$

which is the same as

$$(\mathbf{a+b})(\mathbf{a+b}) = \mathbf{a^2 + 2ab + b^2}. \tag{1.30}$$

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

$$\begin{aligned}(a+b)(a^2+2ab+b^2) = a^3 + 2a^2b + ab^2 + ba^2 + 2ab^2 + b^3 = \\ a^3 + 2a^2b + ba^2 + ab^2 + 2ab^2 + b^3 = \\ \mathbf{a^3 + 3a^2b + 3ab^2 + b^3}\end{aligned} \tag{1.31}$$

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

$$\begin{aligned}(a+b)(a^3+3a^2b+3ab^2+b^3) = \\ a^4 + 3a^3b + 3a^2b^2 + ab^3 + ba^3 + 3a^2b^2 + 3ab^3 + b^4 = \\ a^4 + 3a^3b + ba^3 + 3a^2b^2 + 3a^2b^2 + ab^3 + 3ab^3 + b^4 = \\ \mathbf{a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4}\end{aligned} \tag{1.32}$$

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

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

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

$$\dots$$

This, in general, is the binomial theorem:

$$\begin{aligned}(x+y)^n &= \binom{n}{0}x^n y^0 + \binom{n}{1}x^{n-1}y^1 + \dots + \binom{n}{n}x^0 y^n \\ &= \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k}\end{aligned} \tag{1.33}$$

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

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

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

$$(a+b)^{n+1} = \sum_{k=0}^{n+1}\binom{n+1}{k}a^k b^{n+1-k}. \tag{1.34}$$

We start with the simple equation

$$(a+b)^{n+1} = (a+b)^n(a+b) \tag{1.35}$$

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

$$(a+b)^{n+1} = \left(\sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}\right)(a+b). \tag{1.36}$$

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

$$(a+b)^{n+1} = \sum_{k=0}^{n} \binom{n}{k} a^{k+1} b^{n-k} + \sum_{k=0}^{n} \binom{n}{k} a^k b^{n+1-k}. \tag{1.37}$$

The second term looks already quite similar to the case in equation 1.34, 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 simple grammar rules. That this has a very technical, even *tricky* flavour is much more related to the limitations of our mind that does not see through things as simple as numbers, but has to create formal apparatus not to get lost in the dark woods of reasoning.

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

$$(a+b)^{n+1} = \sum_{k=1}^{n+1} \binom{n}{k-1} a^k b^{n+1-k} + \sum_{k=0}^{n} \binom{n}{k} a^k b^{n+1-k}. \tag{1.38}$$

Please confirm for yourself with pencil and paper that the first summation in equations 1.37 and 1.38 is the same:

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

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

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

$$(a+b)^{n+1} = a^{n+1} + \sum_{k=1}^{n} \binom{n}{k-1} a^{k} b^{n+1-k} + \sum_{k=1}^{n} \binom{n}{k} a^{k} b^{n+1-k} + b^{n+1}. \tag{1.39}$$

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

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

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

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

$$(a+b)^{n+1} = a^{n+1} + \sum_{k=1}^{n} \left(\binom{n}{k-1} + \binom{n}{k} \right) a^{k} b^{n+1-k} + b^{n+1}. \tag{1.40}$$

Now you might recognise Pascal's rule given in equation 1.27 above. Indeed, the Almighty Triangle tells us that $\binom{n}{k-1} + \binom{n}{k} = \binom{n+1}{k}$. In other words, we can simplify the equation to

$$(a+b)^{n+1} = a^{n+1} + \sum_{k=1}^{n} \binom{n+1}{k} a^k b^{n+1-k} + b^{n+1}. \tag{1.41}$$

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

$$(a+b)^{n+1} = \sum_{k=0}^{n+1} \binom{n+1}{k} a^k b^{n+1-k} \tag{1.42}$$

and we are done, since – using some mathematical trickery – we have just derived equation 1.34.

Coming back to the question of how to implement binomial coefficients efficiently, we should compare the two alternatives we have already identified as possible candidates, *viz.* equations 1.21 and 1.22, which are repeated here for convenience:

$$\binom{n}{k} = \prod_{j=1}^{k} \frac{n+1-j}{j}, \tag{1.43}$$

$$\binom{n}{k} = \frac{n^{\underline{k}}}{k!}. \tag{1.44}$$

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

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

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

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

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

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

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

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

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

1.8 Combinatorial Problems with Sets

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

mainly during the first half of the 20^{th}, are tired of discussing these issues.

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

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

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

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

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

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

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

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

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

A similar relation is *subset*. A set A is subset of another set B, iff all elements of A are also in B: $\{1\} \subseteq \{1, 2, 3\}$, $\{1, 3\} \subseteq \{1, 2, 3\}$ and also $\{1, 2, 3\} \subseteq \{1, 2, 3\}$. The last case is interesting, because it asserts that every set is subset of itself. To exclude this case and only talk about subsets that are smaller than the set in question, we refer to the

proper or *strict subset*, denoted as $A \subset B$. An important detail of the subset relation is that there is one special set that is subset of any set, *viz.* the *empty set* $\{\}$ that does not contain any element and which is often denoted as $\varnothing$,

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

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

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

```
ps :: (Eq a) => [a] -> [[a]]
ps [] = [[]]
ps (x : xs) = ps xs ++ map (x:) (ps xs)
```

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

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

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

A completely different approach is based on the observation that there are 2^n possible subsets of a set with n elements and this happens to be the number of values one can represent with a binary number of length n, namely the values $0 \ldots n - 1$. 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 people in the world: those who understand binary numbers and those who do not.

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

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

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

To illustrate this, remember that the value of a decimal number is computed as a sum of powers of 10: $1024 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$. The representation of 1024 as powers of two is of course much simpler: $1024 = 1 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + \cdots + 0 \times 2^0$ 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 (cheating on our number type by using floating point numbers):

$$\begin{aligned}&\mathit{natLog} :: \mathit{Natural} \to \mathit{Natural} \to (\mathit{Natural}, \mathit{Natural})\\&\mathit{natLog}\ b\ n = \textbf{let}\ e = \mathit{floor}\ \$\ \mathit{logBase}\ (\mathit{fromIntegral}\ b)\end{aligned}$$

$$(\mathit{fromIntegral}\ n)$$
$$\textbf{in}\ (e, n - b \uparrow e)$$

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

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

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

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:

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

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 $0 \ldots 2^n - 1$, calculates the exponents of the powers of two for each number and then retrieves the elements in the input set that corresponds to the exponents:

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

The function *ps2* 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 at 0, and the number we start with, namely 0. For each number i: if we have reached the last number, we just know the corresponding subset is the input set itself. Otherwise, we map the function $exponent \rightarrow index$ to the result of $binExp$ applied to the current number i. The mapping function, $exp2idx$, uses the list index operator `!!` to get the element of the input list xs at the position x, which is just an exponent. (Note that we have to convert x from $Natural$ to Int, since `!!` expects an Int value.)

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

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

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

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

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

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

Two other important set operations are intersection and difference. The intersection

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

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

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

$$
\begin{aligned}
&union :: (Eq\ a) \Rightarrow [a] \to [a] \to [a] \\
&union\ a\ b = nub\ (a +\!\!+ b)
\end{aligned}
$$

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

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

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

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

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

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

We now can state the set cover problem more formally: We have a set U, called the *universe*, and a set $S = \{s_1, s_2, \ldots, s_n\}$ 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 should run. No real-world heuristics are available for the kernel to find an optimal balance.

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

overall outcome would be better. This *greedy* algorithm will consequently find only a suboptimal solution, *i.e.* $\{1, 2, 3\}$, $\{1, 4\}$ or even $\{1, 2, 4\}$, $\{3, 5\}$ and $\{1, 6\}$. It, hence, needs one set more than the global optimum.

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

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

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

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

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

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

The function *best* simply goes through all elements of the current instance of S. If *best*

arrives at the end of the list, it just returns the previously identified optimum (m, p). Otherwise, for each element of the current set of subsets, it computes the distance and, should the current distance improve on the result, continues with this current optimum, if it does not, it continues with the old parameters.

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

1.9 Stirling Numbers

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

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

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

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

$$
\begin{aligned}
&\mathit{twoPartitions} = \mathit{fltr} \circ \mathit{p2} \\
&\quad \textbf{where } \mathit{p2}\ [\,] \quad\quad = [([\,],[\,])] \\
&\qquad\quad \mathit{p2}\ (x : xs) = [(x : a, b) \mid (a, b) \leftarrow \mathit{p2}\ xs] +\!\!+ \\
&\qquad\qquad\qquad\quad \mathit{fltr}\ [(a, x : b) \mid (a, b) \leftarrow \mathit{p2}\ xs] \\
&\qquad\quad \mathit{fltr} = \mathit{filter}\ (\lambda p \rightarrow \neg\ (\mathit{null}\ (\mathit{fst}\ p) \vee \\
&\qquad\qquad\qquad\qquad\qquad \mathit{null}\ (\mathit{snd}\ p)))
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

the same set of partitions.

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

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

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

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

$\ldots$	i	$i+1$	$\ldots$	j	$j+1$	$\ldots$
$\ldots$	k_1	k_1	$\ldots$	k_2	k_2	$\ldots$

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

We prove by contradiction on restricted growth and assume that this string is possible with both cases, $k_1 < k_2$ or, alternatively, $k_2 < k_1$. Consider the case $k_2 < k_1$. In this case k_2 must appear in a position $p < i$, otherwise, ordering would be violated. But this contradicts the assumption that $k_2 = \{j, j + 1\}$. So, either we violate ordering or k_2 is not shown completely in the diagram above and, then, the subsets are not complementing. Ordering is violated because it implies that any digit a_i in the string is at most $1 + \max a_1, a_2, \ldots, a_{i-1}$. i is either 0, then k_1, per definition, is 0 as well and

no (natural) number is less than 0, hence k_2 cannot be less than k_1; or i is not 0, then there must be a digit $k_0 = k_1 - 1$. If we assume that $k_2 < k_1$, we must assume that $k_1 - 1 < k_2 < k_1$ and, hence, that $0 < k2 < 1$. But that cannot be, since k_2 is still a natural number. □

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

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

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

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

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

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

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

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

$$(0, r) \to [r]$$
$$(q, r) \to r : go\ q$$

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

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

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

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

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

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

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

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

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

We define the growth restriction as follows:

$$
\begin{aligned}
&rGrowth :: [Int] \to Bool \\
&rGrowth\ [\,] \qquad = True \\
&rGrowth\ (x : xs) = go\ x\ xs \\
&\quad \textbf{where}\ go\ \_\ [\,] \qquad = True \\
&\qquad\qquad go\ d\ (z : zs) = \textbf{if}\ z - d > 1\ \textbf{then}\ False \\
&\qquad\qquad\qquad\qquad\quad \textbf{else let}\ d' = max\ d\ z\ \textbf{in}\ go\ d'\ zs
\end{aligned}
$$

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

$$
\begin{aligned}
&hasN :: Int \to [Int] \to Bool \\
&hasN\ b\ r = length\ (nub\ r) \equiv b
\end{aligned}
$$

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

$$
\begin{aligned}
&toRgsN :: Int \to Int \to Int \to (Int, [Int]) \\
&toRgsN\ b = toRgs\ rst\ b \\
&\quad \textbf{where}\ rst\ r = rGrowth\ r \wedge hasN\ b\ r
\end{aligned}
$$

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

$$
\begin{aligned}
&countRgs :: Int \to Int \to [[Int]] \\
&countRgs\ 1\ n\ = [rgs\ 1\ n\ 1] \\
&countRgs\ b\ n\ \ \mid\ b \equiv n \qquad = [[1\,..\,n-1]] \\
&\qquad\qquad\quad \mid\ b > n \qquad = [\,] \\
&\qquad\qquad\quad \mid\ otherwise = go\ 1 \\
&\quad \textbf{where}\ go\ i = \textbf{let}\ (j, r) = toRgsN\ b\ n\ i \\
&\qquad\qquad\qquad\quad \textbf{in if}\ head\ r \not\equiv 0\ \textbf{then}\ [\,]\ \textbf{else}\ r : go\ (j+1)
\end{aligned}
$$

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

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

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

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

This analogy between RGS and partitions is implemented as:

$$
\begin{aligned}
&\mathit{rgs2set} :: (\mathit{Eq}\ a) \Rightarrow [\mathit{Int}] \to [a] \to [[a]] \to [[a]] \\
&\mathit{rgs2set}\ [\,]\ \_\ \mathit{ps} &&= \mathit{ps} \\
&\mathit{rgs2set}\ \_\ [\,]\ \mathit{ps} &&= \mathit{ps} \\
&\mathit{rgs2set}\ (r : \mathit{rs})\ (x : \mathit{xs})\ \mathit{ps} &&= \mathit{rgs2set}\ \mathit{rs}\ \mathit{xs}\ (\mathit{ins}\ r\ x\ \mathit{ps}) \\
&\quad \textbf{where}\ \mathit{ins}\ \_\ \_\ [\,] &&= \bot \\
&\qquad\quad \mathit{ins}\ 0\ p\ (z : \mathit{zs}) &&= (p : z) : \mathit{zs} \\
&\qquad\quad \mathit{ins}\ i\ p\ (z : \mathit{zs}) &&= z : \mathit{ins}\ (i - 1)\ p\ \mathit{zs}
\end{aligned}
$$

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

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

We now can put everything together:

$$
\begin{aligned}
&\mathit{nPartitions} :: (\mathit{Eq}\ a) \Rightarrow \mathit{Int} \to [a] \to [[[a]]] \\
&\mathit{nPartitions}\ \_\ [\,] = [\,] \\
&\mathit{nPartitions}\ 1\ \mathit{xs} = [[\mathit{xs}]] \\
&\mathit{nPartitions}\ k\ \mathit{xs} \mid k \geqslant \mathit{length}\ \mathit{xs} = [[[x] \mid x \leftarrow \mathit{xs}]] \\
&\qquad\qquad\qquad \mid \mathit{otherwise} \quad = \mathit{go}\ (\mathit{countRgs}\ k\ (\mathit{length}\ \mathit{xs})) \\
&\quad \textbf{where}\ \mathit{go}\ [\,] \quad = [\,] \\
&\qquad\quad \mathit{go}\ (r : \mathit{rs}) = \mathit{rgs2set}\ r\ \mathit{xs}\ (\mathit{take}\ k\ (\mathit{repeat}\ [\,])) : \mathit{go}\ \mathit{rs}
\end{aligned}
$$

The function *nPartitions* receives the number of partitions we would like to have, k, and the original set, *xs*. If the set is empty, the result is empty, too. If we just want one

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

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

This result corresponds to the RGS:

001,
010 and
011.

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

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

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

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

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

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

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

$$S(n,k) = \left\{ {n \atop k} \right\}. \tag{1.45}$$

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

$$\left\{ {n \atop k} \right\} = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ k \times \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\} & \text{otherwise.} \end{cases} \tag{1.46}$$

This translates into Haskell as:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$s(n,k) = \begin{bmatrix} n \\ k \end{bmatrix} \tag{1.47}$$

and can be calculated as

$$\begin{bmatrix} n \\ k \end{bmatrix} = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ (n-1) \times \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} & \text{otherwise.} \end{cases} \tag{1.48}$$

In Haskell, this would be:

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

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

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

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

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

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

1.10 Eulerian Numbers

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

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

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

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

$$
\begin{aligned}
&\mathit{merge} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \to [a] \\
&\mathit{merge}\ [\,]\ \mathit{xs} = \mathit{xs} \\
&\mathit{merge}\ \mathit{xs}\ [\,] = \mathit{xs} \\
&\mathit{merge}\ (x : \mathit{xs})\ (y : \mathit{ys}) \mid x \leqslant y \quad\ = x : \mathit{merge}\ \mathit{xs}\ (y : \mathit{ys}) \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = y : \mathit{merge}\ (x : \mathit{xs})\ \mathit{ys}
\end{aligned}
$$

We first treat the base cases where one of the lists is empty, just yielding the respective other list. Then we compare the heads of the lists. The smaller one goes to the head of the merged list and we recurse with what remains.

We now want to use *merge* on an unordered, that is to say, incompletely ordered list. Therefore, we split the input into a list of ordered sublists. Ordered sublists are often referred to as *run*s. The positions where one ordered list ends and another one begins are called *stepdown*s, reflecting the idea that the sequence of increasing elements is interrupted by stepping down to a smaller element. Here is a Haskell function that splits a list into runs:

$$
\begin{aligned}
&\mathit{runs} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [[a]] \\
&\mathit{runs}\ [\,] \quad\ = [\,] \\
&\mathit{runs}\ \mathit{xs} \quad = \mathbf{let}\ (r, \mathit{zs}) = \mathit{run}\ \mathit{xs}\ \mathbf{in}\ r : \mathit{runs}\ \mathit{zs} \\
&\quad \mathbf{where}\ \mathit{run}\ [\,] \quad = ([\,], [\,]) \\
&\qquad\qquad \mathit{run}\ [x] = ([x], [\,]) \\
&\qquad\qquad \mathit{run}\ (x : y : \mathit{zs}) \mid x > y \quad\ = ([x], y : \mathit{zs}) \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathbf{let}\ (r, \mathit{ys}) = \mathit{run}\ (y : \mathit{zs})\ \mathbf{in}\ (x : r, \mathit{ys})
\end{aligned}
$$

The function applied to the empty list just yields the empty list. Applied to a non-empty list is calls the helper function *run* that returns a tuple consisting of two lists of as. The first element is then the head of the list resulting from *runs* applied to the second list.

The helper function *run* applied on the empty list yields a tuple of twice the empty list. Applied to a list containing only one element, it yield a tuple containing this list and the empty list. Otherwise, the first two elements of the list, x and y, are compared. If $x > y$, x goes to the first list of the resulting tuple, y goes to the rest of the list. This is a stepdown: the first element x is greater than its successor y and, hence, the natural order of the sequence is violated. Otherwise, we continue with $y : zs$ and insert x as the head of the resulting r, the first of the tuple. This is the case, where the run continues.

Let us look at an example: the permutation we already used above $1, 3, 4, 2, 5$. When we call *run* for the first time, we have:

$\mathit{run}\ (1 : 3 : [4, 2, 5])$

and we enter the *otherwise* alternative with $x = 1$:

$\mathit{run}\ (3 : 4 : [2, 5])$.

We again enter *otherwise*, this time $x = 3$:

run $(4 : 2 : [5])$.

But this time we have $4 > 2$ and enter the first branch, that is we yield

$([4], 2 : [5])$.

Going backwards, we see:

$(3 : [4], [2, 5])$
$(1 : [3, 4], [2, 5])$,

which finally appears as result of the first call to *run* in *runs*, which leads to $[1, 3, 4] :$ *runs* $[2, 5]$. $[2, 5]$ is now handled in the same way and we obtain the overall result $[[1, 3, 4], [2, 5]]$. This way, thee entire list is split into two runs.

When we look at the other example, the permutation that was closer to disorder, there are more runs: $5|24|3|1$, which is the proper mathematical notation for the list $[[5], [2, 4], [3], [1]]$ consisting of four runs. If we reverse the list, however, the number of runs reduces and we get the list with only two runs above.

Can we exploit the fact that we can reduce the number of runs by reverting the list? Yes, of course, otherwise the question would not have been asked. It turns out that, for lists with unique elements, if the number of runs r is greater than $\frac{n}{2} + 1$, then r', the number of runs of the reversed list is less than r. In other words, we could check the number of runs, before we start to sort, and revert the list if the number of runs is greater than the half of the length of the list plus 1.

Note that in the real world we often see lists with repeated elements. The repetitions, however, would not spoil the result, since repetitions would just reduce the possible number of runs. In consequence, it may happen that reverting the list, even though the number of runs is less than the half of the list size plus 1, would improve performance. But without reverting, the algorithm is still good, *viz.* comparable to the performance of a cousin of the same size without repetitions.

Let us look at an implementation of *mergesort* that exploits order in the input in this sense. First, we need a function that merges the runs, that is a merge for a list of lists. Let us call this function *multimerge*:

$$
\begin{aligned}
&\mathit{multimerge} :: (\mathit{Ord}\ a) \Rightarrow [[a]] \to [a] \\
&\mathit{multimerge}\ [\,] \qquad\quad = [\,] \\
&\mathit{multimerge}\ [xs] \qquad = xs \\
&\mathit{multimerge}\ (x : y : zs) = \mathit{merge}\ (\mathit{merge}\ x\ y)\ (\mathit{multimerge}\ zs)
\end{aligned}
$$

The function reduces a list of lists of a to a plain list of a. For the empty list, it yields the empty list. For a list that contains only one single list, it yields just that list. For a list with more elements, it merges the first two lists and merges the resulting list with the list that results from *multimerging* the rest. With an example that will become clearer. But let us not use the list with two runs, because that would not let us see the

recursion on *multimerge*. Instead, we use the non-optimal list with four runs. We would start with:

multimerge $[[5] : [2, 4] : [[3], [1]]$

and perform

merge (*merge* $[5]$ $[2, 4]$) (*multimerge* $[[3], [1]]$),

which is

merge $[2, 4, 5]$ (*multimerge* $[[3], [1]]$)

and results in the call to *multimerge*:

multimerge $[[3], [1]]$.

This reduces to

merge (*merge* $[3]$ $[1]$) (*multimerge* $[]$),

which is

merge $[1, 3]$ $[]$,

which, in its turn, is $[1, 3]$. Going backwards, we obtain

merge $[2, 4, 5]$ $[1, 3]$,

which is $[1, 2, 3, 4, 5]$.

We can now put everything together:

```
mergesort :: (Ord a) ⇒ [a] → [a]
mergesort l = let   rs = runs l
                    n' = length l
                    n  = if even n' then n' else n' + 1
              in if length rs > (n `div` 2) + 1
                 then multimerge $ runs (reverse l)
                 else multimerge rs
```

We first create the runs for the input list l. We then compute the length of l n'. If n' is not even, we add one, just to be sure, we later refer to at least the half of n'. Then, if the length of the runs is more than half of n plus 1, then we run *multimerge* on the runs of the reversed input list, otherwise, we run *multimerge* on the runs of l.

There is a well-known mathematical concept that is closely related to the conept of runs: the *Eulerian numbers*, which, as the Stirling numbers, come in two flavours ingeniously called Eulerian numbers of the first kind and Eulerian numbers of the second kind.

The Eulerian numbers of the first kind, denoted by $\left\langle {n \atop m} \right\rangle$, count the number of permu-

tations of a set of n distinct elements where m numbers are greater than the previous element. They, hence, do not count the number of runs directly. They, first, use the number of elements that are actually member of a run without the first element of that run (which is a stepdown). This, actually, is the value of m, which counts, in other words, how many of the elements are not stepdowns. They, second, count the number of permutations that have m elements in such a configuration. For instance, the set $\{1, 2, 3, 4, 5\}$ has $5! = 120$ permutations. There is exactly one permutation where 4 elements are greater than their predecessor, namely the permutation $1, 2, 3, 4, 5$. Hence: $\left\langle {5 \atop 4} \right\rangle = 1$. There is also only one permutation with no element greater than its predecessor, namely $5, 4, 3, 2, 1$. Hence: $\left\langle {5 \atop 0} \right\rangle = 1$. There are 26 permutations with only 3 elements greater than their predecessor and 66 with only 2 elements greater than their predecessor.

You perhaps see immediately that Eulerian numbers can be used to compute the average running time of *mergesort* for input of a given size n. The Eulerian numbers allow us to compute the probability of that input having 1 run, 2 runs, ..., n runs. For the example above, we have the probabiliy $\frac{1}{5!=120}$ that there is no stepdown at all and that we, hence, do not have to do anything; we have the probability $\frac{26}{120} = \frac{13}{60}$ that we have to do only one merge step and the probability of $\frac{66}{120} = \frac{11}{20}$ that we have to do two merge steps and so on. (Knuth provides an extensive analysis in the third volume of his masterpiece.)

Let us look at a smaller set, where we can actually look at all permutations, say $\{1, 2, 3\}$. There is of course 1 permutation for 2 non-stepdowns and also 1 for no non-stepdown at all: $\left\langle {3 \atop 0} \right\rangle = \left\langle {3 \atop 2} \right\rangle = 1$. For 1 element greater than its predecessor, there are $\left\langle {3 \atop 1} \right\rangle = 4$ possibilities, namley: $213, 231, 132, 312$. The sum of all values is of course the number of all permutations, *i.e.* $3! = 6$: $\left\langle {3 \atop 0} \right\rangle + \left\langle {3 \atop 1} \right\rangle + \left\langle {3 \atop 2} \right\rangle = 1 + 4 + 1 = 6$.

Here are some values arranged in – oh, you guessed it already:

1						1							
2					1		1						
3				1		4		1					
4			1		11		11		1				
5		1		26		66		26		1			
6	1		57		302		302		57		1		
7	1	120		1191		2416		1191		120		1	

Make sure yourself that, for each line n, the following identity holds:

$$n! = \sum_{m=0}^{n-1} \left\langle {n \atop m} \right\rangle \tag{1.49}$$

There is a recursive formula to compute the Eulerian numbers, which is

$$\left\langle {n \atop m} \right\rangle = (n-m)\left\langle {n-1 \atop m-1} \right\rangle + (m+1)\left\langle {n-1 \atop m} \right\rangle, \tag{1.50}$$

with $\left\langle {0 \atop m} \right\rangle = \left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1$. In Haskell, this can be implemented as:

```
eulerian1 :: Natural → Natural → Natural
eulerian1 0 _  = 1
eulerian1 _ 0  = 1
eulerian1 n m | m ≡ n − 1 = 1
              | otherwise  = (n − m) ∗ eulerian1 (n − 1) (m − 1)
                           + (m + 1) ∗ eulerian1 (n − 1) m
```

There is also a closed form to compute the nth Eulerian number and this closed form reveals a relation of the Eulerian numbers with the binomial coefficients:

$$\left\langle {n \atop m} \right\rangle = \sum_{k=0}^{m} (-1)^k \binom{n+1}{k} (m+1-k)^n. \tag{1.51}$$

This expands into a series where the results of the products $\binom{n+1}{k}(m+1-k)^n$ are alternately added or subtracted. For instance for $\left\langle {5 \atop 2} \right\rangle$:

$$\left\langle {5 \atop 2} \right\rangle = (-1)^0 \times \binom{6}{0} \times (2+1-0)^5 + (-1)^1 \times \binom{6}{1} \times (2+1-1)^5 + (-1)^2 \times \binom{6}{2} \times (2+1-2)^5, \tag{1.52}$$

which is

$$\begin{array}{rrcrcrcr} & 1 & \times & 3^5 & & & = & 243 \\ - & 6 & \times & 2^5 & = & 6 \times 32 & = & 192 \\ + & 15 & \times & 1^5 & & & = & 15, \end{array}$$

hence: $243 - 192 + 15 = 66$.

It perhaps helps to get the closed form right to look at it in Haskell:

```
eu1closed :: Natural → Natural → Natural
eu1closed n m = go 0
  where go k = let a = (−1) ↑ k
                   b = choose (n + 1) k
                   c = (m + 1 − k) ↑ n
               in (a ∗ b ∗ c) + if k ≡ m then 0
                                         else go (k + 1)
```

The Eulerian numbers of the second kind are as well quite interesting. They deal with *multisets*, that is sets with repeated elements and, as such, they may pave the way to a theory for real world input. In concrete, they count the number of permutations of a multiset with n different elements with m ascents, where an ascent occurs whenever a number is greater than its predecessor. The concept of an ascent, hence, is stronger than that of a non-stepdown. A non-stepdown would include a number that equals its predecessor, but that is not an ascent.

To understand Eulerian numbers of the second kind, we first have to understand how many permutations there are for multisets. It is not the factorial, but the *doublefactorial*, also called the *semifactorial*, of $2n-1$, where n is the number of unique elements in the multiset. For instance, the multiset $\{1,1,2,2,3,3\}$ has $n=3$ unique elements, namely 1, 2 and 3. The doublefactorial of $2n-1=5$, denoted by $n!!$, is 15.

The doublefactorial of n is the product of all numbers $1\ldots n$ that have the same parity as n. If n is even, we multiply all even numbers $2\ldots n$, otherwise, if n is odd, we multiply the odd numbers $1\ldots n$. In Haskell, this may look like this:

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

For instance, $5!! = 1 \times 3 \times 5 = 15$ and $6!! = 2 \times 4 \times 6 = 48$.

The permutations of the multiset $\{1,1,2,2,3,3\}$ are

332211,
221133, 221331, 223311, 233211, 113322, 133221, 331122, 331221,
112233, 122133, 112332, 123321, 133122, 122331.

In the first line, there is one permutation with no ascent, in the second line, there are 8 permutations with two ascents and in the third line, there are 6 permutations with two ascents. The Eulerian numbers of the second kind, denoted by $\left\langle\!\left\langle {n \atop m} \right\rangle\!\right\rangle$, for a multiset with $n=3$ different elements, thus, are: $\left\langle\!\left\langle {3 \atop 0} \right\rangle\!\right\rangle = 1$, $\left\langle\!\left\langle {3 \atop 1} \right\rangle\!\right\rangle = 8$ and $\left\langle\!\left\langle {3 \atop 2} \right\rangle\!\right\rangle = 6$. Here are some values arranged in the last triangle you will see for some time:

1							1							
2						1		2						
3					1		8		6					
4				1		22		58		24				
5			1		52		328		444		120			
6		1		114		1452		4400		3708		720		
7	1		240		5610		32120		58140		33984		5040	

Again, this triangle shows some interesting properties. The sum of each row n is of course the doublefactorial of $2n-1$, *e.g.* $\left\langle\!\left\langle {3 \atop 0} \right\rangle\!\right\rangle + \left\langle\!\left\langle {3 \atop 1} \right\rangle\!\right\rangle + \left\langle\!\left\langle {3 \atop 2} \right\rangle\!\right\rangle = 5!! = 15$ and $\left\langle\!\left\langle {4 \atop 0} \right\rangle\!\right\rangle + \left\langle\!\left\langle {4 \atop 1} \right\rangle\!\right\rangle +$

$\left\langle\!\left\langle {4 \atop 2} \right\rangle\!\right\rangle + \left\langle\!\left\langle {4 \atop 3} \right\rangle\!\right\rangle = 7!! = 105$. Neatly, the oughter right-hand diagonal shows the factorials. The last value of each row, $\left\langle\!\left\langle {n \atop n-1} \right\rangle\!\right\rangle$, hence, is $n!$.

Here comes the formula to compute the Eulerian numbers of the second kind recursively:

$$\left\langle\!\left\langle {n \atop m} \right\rangle\!\right\rangle = (2n - m - 1) \left\langle\!\left\langle {n-1 \atop m-1} \right\rangle\!\right\rangle + (m+1) \left\langle\!\left\langle {n-1 \atop m} \right\rangle\!\right\rangle, \tag{1.53}$$

with $\left\langle\!\left\langle {0 \atop 0} \right\rangle\!\right\rangle = 1$ and $\left\langle\!\left\langle {0 \atop m} \right\rangle\!\right\rangle = 0$. In Haskell this is:

```
eulerian2 :: Natural → Natural → Natural
eulerian2 0 0  = 1
eulerian2 0 _  = 0
eulerian2 n m = (2 * n - m - 1) * eulerian2 (n - 1) (m - 1)
              + (m + 1)         * eulerian2 (n - 1) m
```

The Eulerian numbers, as many other things in mathematics, are named after Leonhard Euler (1707 – 1783), a Swiss mathematician and one of the most important mathematicians of all time. He is certainly the most important mathematician of his own time, and, for sure, the most productive one ever. He turned his family into a kind of math factory that produced thousands of papers in number theory, analysis, mechanics, optics, astronomy, ballistics and even music theory. He is also regarded as the founder of graph theory and topology. Modern terminology and notation is strongly based on Euler. He proved many theorems and made even more conjectures. But he was also a great math teacher, as his *Letters to a German Princess* show in which he lectured on mathematical subjects to non-mathematicians.