Contents

1 Primes

1.1 Gaps

The natural numbers are very simple in the sense that there are very simple ways to enumerate them all (given that we have infinite time and patience to do so, of course). Given a starting point, such as 0, 1 or any other number, we can generate all numbers from this point on just by counting up. This fact is so obvious that it appears ridiculous to visualise it like this:

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

A formula to generate this sequence might be: $f(n) = f(n-1) + 1$, *i.e.* the value for n equals the value of $n-1$ plus 1, which can be simplified to the closed form n. In fact, the function $f(n-1)+1$, starting with $n = 0$ is just the definition of natural numbers. The number used for counting, 1, is therefore called *unity*.

A slightly more interesting sequence is $f(n) = f(n-1) + 2$, that is is the multiplication table of 2, *i.e.* $f(n) = 2 \times n$:

1	2	3	4	5	6	7	8	9	10	...
2	4	6	8	10	12	14	16	18	20	...

This table shows exactly half of all numbers, *viz.* the even numbers. So, let us look at the second half of the numbers, the odd ones. The following table, correspondingly, shows a third of all numbers, namely those, divisible by 3:

1	2	3	4	5	6	7	8	9	10	...
3	6	9	12	15	18	21	24	27	30	...

Every second number in this table is even and, hence, already appears in the previous table. The logically next table, the one containing multiples of 4, would contain a quarter of all numbers. But it is not very interesting, since all numbers in that table already appear in the multiplication table for 2. With 5, however, we could get some new numbers to fill up the second half:

1	2	3	4	5	6	7	8	9	10	...
5	10	15	20	25	30	35	40	45	50	...

We see that every second number was already in the multiplication table for 2 and every

third number in that for 3. So, it seems it is not too easy to get all numbers together – there is a lot of repetition in multiplication tables!

We can safely jump over the logically next table, 6, because all numbers in that table are already in the second table and, since 6 is a multiple of 3, in the third table as well. In the hope to find some more numbers of the second half, we continue with 7:

1	2	3	4	5	6	7	8	9	10	...
7	14	21	28	35	42	49	56	63	70	...

Every second number is in the table for 2, every third number appears also in the third table and every fifth number appears in the table for 5. The first new number we see is $7 \times 7 = 49$. It appears that we are running short of novelties! Indeed, we now have to skip 8 (since it is even), 9 (since it is a multiple of 3) and 10 (since it is not only even, but also a multiple of 5). The first number with some potential to bring something new is 11:

1	2	3	4	5	6	7	8	9	10	...
11	22	33	44	55	66	77	88	99	110	...

With some disappointment, we have to admit that there is no number up to $n = 10$ that we have not seen so far (besides 11 itself). With 12 we will not have more luck, since 12 is even. So let us have a look at 13 before we give up:

1	2	3	4	5	6	7	8	9	10	...
13	26	39	52	65	78	91	104	117	130	...

So, filling up the second half of the numbers does not appear to be an easy task. Very few numbers are really "new" in the sense that they are not multiples of numbers we have already seen. On second thought, this fact is not so curious anymore. The numbers that appear in a table for k have the form $n \times k$ and, for all $n < k$, we, of course, have seen n already in the tables for n, since $n \times k$ is the same as $k \times n$. In the table for 7, 7×7 was therefore the first number we had not yet seen. For numbers greater than k, the same is true for all multiples of earlier numbers; so 8×7, for instance, appears in the multiplication table of 4, since 8 is a multiple of 4. 9×7 appears in the table for 3, since 9 is a multiple of 3 and so on.

The really curious fact in this light is another one: that there, at all, are numbers that do not appear in tables seen so far. In fact, the numbers 2, 3, 5, 7, 11 and 13 never appear in any table, but their own. For 2 and 3, this is obvious, because 2 is the number we are starting with, so there simply is no table of a smaller number in which 2 could appear. Since 3, 5, 7 and so on are all odd, they cannot appear in the table for 2. But could they not appear in a later table? 5, obviously, cannot appear in the table for 3, since 5 is not a multiple of 3. The same holds for 7 and 7 is not a multiple of 5 either, so it will not appear in that table. We can go on this way with 11, 13 and any other number not seen so far and will always conclude that it cannot have appeared in an earlier table, since it is not a multiple of any number we have looked at until now.

We may think that this is a curiosity of small numbers until, say, 10 or so. But, when we go further, we always find another one: 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47. These unreachable numbers are called *primes* as opposed to *composites*, which are composed of other numbers in terms of multiplication. A prime, by contrast, is a number that is divisible only by 1 and itself.

Until now, we did not make a great effort to list prime numbers. In fact, almost every second number so far was prime. But it is hard to predict the next prime for any given n. For instance, what is the next prime after 47? Since, $48 = 2 \times 2 \times 2 \times 2 \times 3$, $49 = 7 \times 7$, $50 = 2 \times 5 \times 5$, $51 = 3 \times 17$ and $52 = 2 \times 2 \times 13$ none of these numbers is prime. The next one is only 53. So, what is the next prime after 6053 (which itself is prime)? Is there any more prime at all after this one or any other prime greater than the last prime we found so far in general? Let us examine how to find primes and which number is the last prime.

1.2 Finding Primes

In the previous section, we have already adopted a classical method to find primes, namely a *sieve*. We started by picking 2 and generated all multiples of 2. We then took the first number greater than 2 that was not in the list of multiples of 2, 3, and generated all its multiples. We continued by picking the first number greater than 3 that was neither in the list of multiples of 2 nor in that of multiples of 3, *viz.* 5. As a result, we identify primes as the heads of these list, $2, 3, 5, 7, 11, 13, \ldots$.

This method was invented by Eratosthenes, a polymath of the 3^{rd} century BC, who was librarian of the library of Alexandria, which we already visited discussing Euclid. Eratosthenes wrote books on history, poetry, music, sports, math and other topics; he is considered founder of geography and he estimated the circumference of the earth remarkably close to the accurate value of about 40 000km known today. The exact precision of his estimate is subject to dispute – scholars refer to values he may have given between 39 500km, which would correspond to an error of only 1.5%, and 46 500km, which would be a more likely deviation of about 15%.

The *Sieve of Eratosthenes* goes as follows: write all numbers from 2 up to n, the upper limit of the range, for which you want to calculate the prime numbers. Eliminate all multiples of 2. Take the first number greater than 2 that was not yet eliminated and eliminate all its multiples. Take the first number greater than that number and proceed as before until the next number is n.

The following code shows an implementation in Haskell without upper limit, *i.e.* it finds all prime numbers exploiting lazy evaluation:

$$\begin{aligned} &\mathit{erato} :: [\mathit{Natural}] \\ &\mathit{erato} = \mathit{sieve}\ 2\ [2\,..] \end{aligned}$$

$$\begin{array}{l}\textbf{where } \mathit{sieve}\ x\ xs = \textbf{case } \mathit{filter}\ (\lambda p \to p\ `\mathit{rem}`\ x \not\equiv 0)\ xs \textbf{ of} \\ \qquad\qquad\qquad\qquad\qquad\qquad [\,] \to [x] \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{ps} \to x : \mathit{sieve}\ (\mathit{head}\ \mathit{ps})\ \mathit{ps}\end{array}$$

The function *sieve* has two arguments: the current number (starting with 2) and the list of all numbers starting with 2. The function filters all numbers out that leave remainder 0 divided by 2. If we have reached the last number, *i.e.* there are no more numbers in the list of all numbers, we just return the current number (which to know, if this case ever manifests, would be of the utmost interest). Otherwise, we insert the current number as head of the recursion on *sieve* that takes the head of the filtered list as the current number and the filtered list itself. In the first round we would get:

$\mathit{sieve}\ 2\ [2,3,4,5,6,7,8,9,...] = 2 : \mathit{sieve}\ 3\ [3,5,7,9,...]$

and continue with:

$\mathit{sieve}\ 3\ [3,5,7,9,...] = 3 : \mathit{sieve}\ 5\ [5,7,...]$
$\mathit{sieve}\ 5\ [7,...] = 5 : \mathit{sieve}\ 7\ [...]$

and so on.

If you call *erato* just like that, the function goes on forever, that is until the resources of your computer are exhausted or you interrupt the program. A call like *take n erato* would show the first n primes; a call like *takeWhile* $(<n)$ *erato* would show all primes less than n; be careful: a call like *takeWhile* $(\not\equiv n)$ *erato*, where n is not itself a prime, will run forever!

The principal merit of *erato* is its simplicity and conciseness; it is not very efficient however. Much more efficient is a very nice variation of the sieve of Eratosthenes given in the *Haskell Road*. This implementation is based on a primality test to decide whether a number enters the list of primes or not. The test itself uses a list of primes:

$$\begin{array}{l}\mathit{nextPrime} :: [\mathit{Natural}] \to \mathit{Natural} \to \mathit{Natural} \\ \mathit{nextPrime}\ [\,]\ n \qquad = n \\ \mathit{nextPrime}\ (p : \mathit{ps})\ n \mid \mathit{rem}\ n\ p \equiv 0 = p \\ \qquad\qquad\qquad\qquad\quad \mid p \uparrow 2 > n \quad = n \\ \qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \quad = \mathit{nextPrime}\ \mathit{ps}\ n\end{array}$$

nextPrime receives a list of primes and a number, n, to be tested for primality; if n is a prime, this number is returned, otherwise, the first prime dividing that number is returned.

If the first prime in the list, p, divides n, then p is returned; otherwise, if n is less than p^2, n is returned. Since p does not divide n and n is smaller than p^2 and p is the smallest prime remaining in the list, there will be no two primes in the list that multiplied with each other yield n. Because the primes remaining in the list are all greater than p, any

product of two of them will obviously be greater than p^2 and, hence, greater than n. Therefore, n must be a prime. Otherwise, if n is greater than p^2, we continue the search with the next prime in the list.

The following code sequence turns *nextPrime* into a Boolean primality test:

$$
\begin{aligned}
&\mathit{prime} :: \mathit{Natural} \to \mathit{Bool} \\
&\mathit{prime}\ n \mid n \equiv 1 \quad\ = \mathit{False} \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{ldp}\ n \equiv n \\
&\mathit{ldp} :: \mathit{Natural} \to \mathit{Natural} \\
&\mathit{ldp} = \mathit{nextPrime}\ \mathit{allprimes}
\end{aligned}
$$

ldp stands for *least dividing prime* and yields the first prime number that divides n. It calls *nextPrime* with a list of all primes. The function is used in the test function *prime*, which compares n with *ldp* n, *i.e.* if the first prime that divides n is n, then n is a prime itself.

Now, where does the list of all primes come from? This is the beautiful part of the code. It is created in terms of *prime* used as a filter on the natural numbers:

$$
\begin{aligned}
&\mathit{allprimes} :: [\mathit{Natural}] \\
&\mathit{allprimes} = 2 : \mathit{filter}\ \mathit{prime}\ [3\,..]
\end{aligned}
$$

Note that it is essential here to add 2 explicitly to the result, since it is 2 that bootstraps the algorithm. If we created *allprimes* as *filter prime* $[2\,..]$, we would introduce an infinite regress: *allprimes* would try to filter prime numbers using *prime*, which, in its turn, uses *nextPrime* with *allprimes*, which, again calls *prime* to get a prime out of the list of numbers. With 2 already in that list, *prime* will first test primality of n, which is 3 in the first round, with the head of *allprimes*, *i.e.* 2. Since 2 does not divide 3 and $2^2 > 3$, 3 is returned and the algorithm is up and running.

A lot of sieves have been developed since Eratosthenes and many of them are much more efficient than Eratosthenes' sieve. A particular interesting one is the *Sieve of Sundaram*, which was developed by an Indian math student in the 1930ies. Sundaram's sieve finds the odd primes up to a limit of $2n + 2$ with the minor drawback that 2 is not in the list. Since 2 is the only even prime, this issue is easily solved by just adding 2 explicitly.

The algorithm is based on the fact that odd composites have odd factors. As we have already seen in the previous chapter, odd numbers can be represented as $2n + 1$. Odd composites, therefore, have factors of the form $(2i+1)(2j+1)$. If we multiply this out, we obtain $4ij+2i+2j+1$. We can split this sum into two terms of the form $(4ij+2i+2j)+1$ using the associative law. We move 2 out of the first term yielding $2(2ij + i + j) + 1$. Sundaram's algorithm cleverly removes all numbers of the form $2ij + i + j$ from the list of all numbers up to a given limit, doubles the remaining numbers and adds 1 to the result. Since all resulting numbers are again of the form $2n + 1$, they are all odd and,

since all numbers of the form $2ij + i + j$ have been removed, we know that none of the resulting odd numbers $2n + 1$ is composite.

Here is a possible implementation:

```
sund :: Natural → [Natural]
sund n = 2 : [2 * x + 1 | x ← [1 .. n] \\
                          [i + j + 2 * i * j | i ← [1 .. lim 0],
                                               j ← [i .. lim i]]]
  where lim 0 = floor $ sqrt (fromIntegral n / 2)
        lim i = floor $ fromIntegral (n − i) / fromIntegral (2 * i + 1)
```

We create the list of all numbers $[1 .. n]$ and subtract from it the list of all numbers of the form $i + j + 2ij$. the numbers i and j are generated as $[1 .. lim\ 0]$ and $[i .. lim\ i]$ respectively.

With the limits *lim*, we avoid multiplying pairs of numbers twice, such as 2×3 and 3×2. To achieve this, we generate i starting from 1 up to the greatest whole number less than the square root of half of n. The reasoning for this limit is that we do not want to generate too many is beyond n, since, at the end, we want to have primes only up to $2n + 2$, *i.e.* $i + j + 2ij <= n$ (since, at the end, we still multiply by 2). The smallest j we will use is i, which, injected into the above formula, yields $2i^2 + 2i$. For huge is, the second part is negligible, so instead of using this formula, we simplify it to $2i^2$ and generate is from 1 to a number x that squared and duplicated is at most eqal to n. This number, obviously, is the square root of half of n.

For j, the lower limit is i; the upper limit, $i + j + 2ij <= n$, can be expressed in terms of n and i: First we bring i on the other side: $j + 2ij <= n - i$; now we factor j out: $j(1 + 2i) <= n - i$, divide by $1 + 2i$: $j <= \frac{n-i}{2i+1}$ and get the limit defined in the code as *lim i*.

A bit confusing might be that we use *lim* 0 to calculate the limit for i. This is just a trick to use the same function for i and j. In fact, the limit for i is a constant relative to n. We could have it defined without an argument at all. But this way, using the same function for i and j, it looks nicer.

Sieves do a great job in creating lists of prime numbers. They are weak, when it comes to finding new prime numbers. Since sieves depend on primes discovered so far, any search for new prime numbers must start at the beginning, *i.e.* with 2. How far we will get, depends on available time and computing power. Those are serious limits in finding new primes, which may lie far ahead on the number ray. Instead of going forward step by step, as sieves do, we might want to make huge leaps forward ignoring thousands and millions of numbers.

A very simple method to guess a prime is based on the observation that primes often come in pairs. This, obviously, does not introduce huge leaps leaving thousands and

millions of numbers out, it just leaves one number out. For instance 3 is prime and $3+2=5$ is prime too. Now, 5 is prime and $5+2=7$ is prime as well. 11 is prime and $11+2=13$ is prime too. So are 17 and 19, 29 and 31, 41 and 43, 59 and 61 and 71 and 73. But there are also many primes without a twin, *e.g.* 37, 53, 67, 83 and 89. In fact, if all primes came as twins, every second number would be prime and that, definitely, is not true. How many prime pairs there are and whether there are infinitely many of them is not known today. It is an unresolved problem in mathematics.

Bigger leaps are introduced by so called *Mersenne primes*, which have the form 2^n-1. This method of finding primes is based on the fact that many primes are powers of 2 minus 1, for instance: $3=2^2-1, 7=2^3-1, 31=2^5-1, 127=2^7-1$. Not all powers of 2, however, lead to Mersenne primes. $2^4-1=15$ is the composite number 3×5. $2^6-1=63$ is composite as well. So are $2^8-1=255=3\times 5\times 17, 2^9-1=511=7\times 73, 2^{10}-1=1023=3\times 11\times 31$. It turns out that all numbers of the form 2^n-1, where n is composite, are composite numbers as well. Mersenne primes are hence restricted to 2^p-1 with p a prime number. But even under this condition, not all Mersenne numbers are indeed primes, for instance $2^{11}-1=2047=23\times 89$. It is not known how many Mersenne primes there are and if there are infinitely many of them. This, again, is still an open problem in mathematics.

Most huge primes found today are Mersenne Primes. The search is assisted by the *Great Internet Mersenne Prime* Search (GIMP), which already found more than a dozen primes, most of them are also the greatest prime numbers known so far. The greatest of them is more than 10 million digits long.

Marin Mersenne (1588 – 1648), after whom Mersenne primes are named, was a priest who taught theology and philosophy in France. He wrote on philosopy and theology, but also on math and acoustics and he edited works of acient mathematicians such as Euclid and Archimedes. Mersenne was also a great organiser of science who corresponded with many mathematicians and scientists of his time including René Descartes, Galileo, Pierre Fermat and Etienne Pascal, the father of Blaise.

The main contribution to math is a list of Mersenne primes he compiled up to the exponent 257. There are some flaws in the list, he missed some primes and added some composite numbers. However, the effort is still impressive considering that all the math was done by hand.

Even greater leaps are introduced by searching for *Fermat primes*, named after our friend Pierre Fermat. These primes have the form $2^{2^n}+1$. 3, for example, is a Fermat prime, since $2^{2^0}+1=2^1+1=2+1=1$. 5, as well is a Fermat prime, since $2^{2^1}+1=2^2+1=4+1=5$. The next Fermat prime is $2^{2^2}+1=2^4+1=16+1=17$. The next is $2^{2^3}+1=2^8+1=256+1=257$. The next, as you may have already guessed, is $2^{2^4}+1=2^{16}+1=65536+1=65537$. $2^{2^5}+1=4294967297$, however, is not a prime, since $4294967297=641\times 6700417$. The next one $2^{2^6}+1=18446744073709551617$, as well, is composite, since $18446744073709551617=274177\times 67280421310721$. As with Mersenne primes, we see that not all numbers constructed following the Fermat prime

formula are actually primes. There is just a certain probability (which, hopefully, is greater than that of randomly picking a number) that a Fermat prime is indeed a prime. The largest Fermat prime known today is actually $2^{16}+1$ and there is evidence that the number of Fermat primes is finite.

1.3 The fundamental Theorem of Arithmetic

The theorem with the somewhat bombastic name *fundamental theorem of arithmetic* states that every natural number other than unity either is a prime or can be expressed as a unique product of primes, its *prime factorisation*. "Unique", here, means that, for every natural number, there is exactly one prime factorisation.

On the first sight, this might appear trivial, but consider an example: 24 can be expressed as product in different ways, for instance: $24 = 3 \times 8$ and $24 = 4 \times 6$. 4, 6 and 8, however, are not prime numbers: $4 = 2 \times 2$, $6 = 2 \times 3$ and $8 = 2 \times 2 \times 2$ and, in consequence, the different products reduce to $24 = 2 \times 2 \times 2 \times 3$. In other words, there are many ways to obtain a number by multiplying other numbers, but there is only one way to obtain this number by multiplying prime numbers.

Let us have a look at some prime factorisations: 2 is prime and its prime factorisation is just 2. 3 is prime and its prime factorisation is just 3. 4 is composite and its prime factorisation is $2 \times 2 = 2^2$. Here are the numbers 5 to 20 and their prime factorisation:

$5 = \{5\}$
$6 = \{2, 3\}$
$7 = \{7\}$
$8 = \{2^3\}$
$9 = \{3^2\}$
$10 = \{2, 5\}$
$11 = \{11\}$
$12 = \{2^2, 3\}$
$13 = \{13\}$
$14 = \{2, 7\}$
$15 = \{3, 5\}$
$16 = \{2^4\}$
$17 = \{17\}$
$18 = \{2, 3^2\}$
$19 = \{19\}$
$20 = \{2^2, 5\}$

The facts constituting the fundamental theorem are known since antiquity and are outlined and proven in Book VII of the Elements. Many other proofs have been suggested since approaching the theorem from different angles. Indeed, its name already suggests that it is not an ordinary theorem, but is right in the centre of a whole bunch of problems

in arithmetic.

In this section, we will look at one proof. We will basically establish that 1. every number can be factored into primes and 2. for any number, this factorisation is unique. We will also see, as a corollary to 1, that there are infinintely many primes.

The first theorem – that every number can be factored into primes – is quite simple. We distinguish two cases: the number is either prime or composite. If it is prime, the factorisation is simply that number. Otherwise, if it is not prime, there are two numbers, a and b such that $a \times b = n$, where n is the number in question. Now, a and b either are prime numbers or there are other numbers a_1 and a_2, such that $a_1 \times a_2 = a$, and b_1 and b_2, such that $b_1 \times b_2 = b$. Indeed, that a number is composite means essentially that there exist two other numbers that divide this number. Therefore, every number can be expressed as a product of prime numbers. $\square$

Now we will prove the fundamental theorem of arithmetic. We will use a proof technique called *indirect proof* or *proof by contradiction*. We have already used this type of proof silently and it is in fact a quite common tool in reasoning. The strengths of indirect proofs are that they are often very simple, much simpler than direct proofs, and that they can prove things that we cannot demonstrate. The latter is of major importance, in particular when talking about infinitely big or small things. We can prove, for example, that there are infinitely many primes without the need to construct infinitely many primes.

Indirect proofs also have some drawbacks, though. The most important one is that they do not provide a method to compute the result. An indirect proof may be used to prove the existence of something, but does not provide a method to construct that "something" (such as the result of a given function or the greatest prime number). They are therefore sterile in the sense that we obtain only the abstract knowledge that some theorem is true, but no further insight into the concepts under investigation and no new methods to work with these concepts. That is quite poor and, indeed, many mathematicians have expressed their inconvenience or even disgust when confronted with indirect proofs. There is even a philosophical tradition within mathematics, *mathematical constructivism*, that aims to find direct proofs for all mathematical theorems for which only indirect proofs are known today. Without taking side in the philosophical debate, most mathematicians would agree today that an indirect proof should be the last resort, *i.e.*, when there is a direct proof, it should be preferred.

So, what is an indirect proof in the first place? The proof of a theorem A works by demonstrating that the assumption $\neg A$ leads to a contradiction. We therefore start the proof by stating what we assume to be true. Let us look, as a simple example, a variant of Euclid's proof that there are infinitely many primes:

Assume that there is a finite number of primes. Then we can enumerate the set P of all primes as $P = \{2, 3, 5, \ldots, p\}$, where p is the last prime. The product of the primes in this set is a composite number: $n = 2 \times 3 \times 5 \times \cdots \times p$. So, what about $n + 1$?

This number is either prime, then P was incomplete, which immediately contradicts our assumption; or it is composite and then it has a prime factorisation. But none of the primes in P can be part of that factorisation, because no number greater than 1 divides both n and $n+1$: 2 divides n and $n+2$, but not $n+1$; 3 divides n and $n+3$, but not $n+1$, p divides n and $n+p$, but not $n+1$. Therefore, there must be at least one prime that is not in P, which, again, contradicts our assumption. $\square$

There are, hence, infinitely many primes.

We now prove the fundamental theorem of arithmetic, *i.e.* that there is only one way to factor any given number into primes. We prove this by contradiction and assume that there is at least one number for which it is actually possible to find more than one prime factorisation. We must be very cautious about such assumptions, since we want the contradiction to hit the right place. We, therefore, assume that there is *at least* one number without assuming anything further – there may be just that one number, there may be many or it may be even true for all composites that there is more than one prime factorisation.

In the following, however, we will talk about just one such number. If there is only one number with that property, we talk about that one. Otherwise, if there are many, we talk about the smallest number with that property. We can simply verify for small numbers, in particular 4, the smallest composite number, that there is only one prime factorisation, *i.e.* 2×2. If there is a number with more than one factorisation, it is definitely greater than 4.

We call this smallest number for which more than one prime factorisation exist m:

$$m = p_1 \times p_2 \times \cdots \times p_r \tag{1.1}$$

$$m = q_1 \times q_2 \times \cdots \times q_s \tag{1.2}$$

The ps and qs in these equations are all primes. Also, the ps and qs differ, such that at least one p is not in the list of qs and vice versa. To illustrate this, the ps could be 3 and 8 (if 8 was a prime number) and the qs could be 4 and 6 (if 4 and 6 were prime numbers) in the factorisations of 24. 4, 6 and 8, of course, are not prime numbers. But what we claim (to, hopefully, create a contradiction) is that there are numbers for which different decompositions are possible even with prime numbers.

We further assume that the two factorisations, the ps and qs above are ordered, such that

$$p_1 \leq p_2 \leq \cdots \leq p_r$$

and

$$q_1 \leq q_2 \leq \cdots \leq q_s.$$

Now, p_1 and q_1 must be different, *i.e.* either $p_1 < q_1$ or $q_1 < p_1$, for, if $p_1 = q_1$, we could divide both sides, the p-factorisation and the q-factorisation, by p_1 and obtain a number with two factorisations that is actually smaller than m – but we assume that m is the smallest number with that property. This assumption forces us to also assume that either $p_1 < q_1$ or $q_1 < p_1$. Let us say that p_1 is the smaller one. (It is irrelevant which one it actually is. If we chose q_1 to be the smaller one, we would just swap ps and qs in the following equations.) We can now compute a number m':

$$m' = m - (p_1 \times q_2 \times \cdots \times q_s), \tag{1.3}$$

for which it, obviously, holds that $0 < m' < m$, *i.e.* m' is smaller than m and, hence, has a unique prime factorisation (since m is the smallest number with the property that it has more than one prime factorisation).

Now, by substituting for m, we derive:

$$m' = (p_1 \times p_2 \times \cdots \times p_r) - (p_1 \times q_2 \times \cdots \times q_s) \tag{1.4}$$

and, by factoring p_1 out, we get:

$$m' = p_1 \times (p_2 \times p_3 \times \cdots \times p_r - q_2 \times q_3 \times \cdots \times q_s) \tag{1.5}$$

and clearly see that p_1 is a factor of m'. But we can also derive

$$m' = (q_1 \times q_2 \times \cdots \times q_s) - (p_1 \times q_2 \times \cdots \times q_s), \tag{1.6}$$

from which, by dividing by $q_2 \times q_3 \times \cdots \times q_s$, we can further derive

$$m' = (q_1 - p_1) \times (q_2 \times q_3 \times \cdots \times q_s). \tag{1.7}$$

Since p_1 is a factor of m', it must be a factor of either $q_1 - p_1$ or $q_2 \times q_3 \times \cdots \times q_s$. (Remember that there is only one way to factor m' into primes, since it is smaller than m, the smallest number with more than one prime factorisation.) It cannot be a factor of $q_2 \times q_3 \times \cdots \times q_s$, since all the qs are primes and greater than p_1. So, it must be a factor of $q_1 - p_1$. In other words, there must be a number, say, h for which it holds that

$$q_1 - p_1 = p_1 \times h \tag{1.8}$$

By adding p_1 to both sides we get $q_1 = p_1 \times h + p_1$. By factoring p_1 out on the right-hand side of the equation we obtain:

$$q_1 = p_1 \times (h + 1) \tag{1.9}$$

In other words, p_1 is a factor of q_1. But this is a contradiction, since q_1 is prime. □

1.4 Factoring

In the previous section, we have made heavy use of factoring, *i.e.* of decomposing a number into its prime factors. But we did so without indicating an algorithm. We have just stated that $9 = 3 \times 3$ or $21 = 3 \times 7$. For such small numbers, that is certainly acceptable – we have learnt our multiplication tables in school and can immediately say that $32 = 2^5$. With bigger numbers, this becomes increasingly difficult and, therefore, we clearly need an algorithm.

Unfortunately, no efficient factorisation algorithm is known. In fact, there is one that is sufficiently fast, but that one does not run on traditional computers. It is a quantum algorithm. It was developed by the American mathematician Peter Shor in 1994. It has already been implemented on real quantum computers several times and in 2012 it was used to factor 21. That is not the greatest number factored by a quantum computer so far. With another approach, not involving Shor's algorithm, but a simulation method called *abbiatic quantum computation*, a Chinese team achieved to factor 143 as well in 2012.

We will not enter the quantum world here, but rather stick to classical algorithms, even if this leaves us with highly inefficient programs that need exponential time to factor numbers. There is something good about the fact that factorisation is hard: many algorithms in cryptography are based on it. So, would factorisation be made much simpler overnight, our online banking passwords and other data would not be secure anymore. On the other hand, there is no proof that factorisation will remain a hard problem forever. It is not even known to which *complexity class* factorisation belongs.

Complexity classes, in theoretical computer science, descibe the level of difficulty of solving a problem. There are a lot of complexity classes; for the moment, two are sufficient: P and NP. P stands for *polynomial time*. Polynomial refers to formulas of the form $n^a + c$ or similar, where n is the size of the input (for example the number we want to factor) and a and c are constants or even other – but similar – formulas. Essential is that, in formulas that describe the cost of algorithms that are solutions to P-problems, n does never appear as exponent. Algorithms, whose cost is described by formulas where n appears in exponents, *e.g.* a^n, are exponential. Such algorithm are considered unfeasible for input of relevant size.

The interesting point about NP, now, is that solutions for problems in this class need an incredible amount of steps, such as exponential time where n appears in the exponent, but, if a potential answer is known, it is extremely easy to verify if this answer is correct. The acronym NP means *non-deterministic polynomial time*. The name refers to the fact that if an answer is known it can be verified in polynomial time (the verification, hence, is a P-problem). Where the answer comes from, however, is unclear – it appears out of the blue, in a non-determinist c way. It could be chosen randomly, for instance, or a magus, like Merlin, could have suggested it. Notice that this is definitely a characteristic of factorisation. Given a number such as 1771, it may be hard to say what its prime factors are. If we were told, however, that 7 is one of the factors, we can use division to verify that $1771 \bmod 7 = 0$ and even to reduce the problem to finding the factors of $1771/7 = 253$.

Today, factorisation is not considered to be in NP. Shor's algorithm is a quantum probabilistic algorithm (belonging to class BQP – *bounded-error quantum probabilistic*) and, therefore, it is assumed that it may belong also to a classic probabilistic class (such as BPP – *bounded-error probabilistic polynomial*). This assumption is supported by the fact that there appears to be a consistent distribution of primes – but more on that later.

Factorisation is an area with extensive research. To that effect, there are many algorithms available, most of them exploiting probabilistic in some way or another. We will stick to an extremely simple approach that, basically, uses a trial-and-error method. We simply go through all prime numbers, for this purpose we can use one of the prime number sieves, and try to divide the input number by each one until we find a prime that divides this number. If we do not find such a prime, the number must be prime and its factorisation is just that number.

Here is the searching algorithm:

$$
\begin{aligned}
&\mathit{findf} :: \mathit{Natural} \to \mathit{Natural} \to [\mathit{Natural}] \to (\mathit{Natural}, \mathit{Natural}) \\
&\mathit{findf}\ n\ \_\ [\,] = (1, n) \\
&\mathit{findf}\ n\ l\ (p : ps) \mid p > l \qquad = (1, n) \\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathbf{case}\ n\ `\mathit{quotRem}`\ p\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (q, 0) \to (p, q) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\_, r) \to \mathit{findf}\ n\ l\ ps
\end{aligned}
$$

The function *findf* receives three arguments: the input number, n, an upper limit, l, and the list of primes. It yields a pair of numbers: the first is the prime number that divides n, the second is the quotient of n divided by the prime. If the list of primes is exhausted, a case that, as we know from the previous section, is extremely rare, we just return $(1, n)$ to indicate that we have not found a proper solution. Otherwise, we check if we have reached the upper limit. In this case, we again yield $(1, n)$ to signal that no proper solution was found. Otherwise, we divide n by the first prime in the list and, if the remainder is zero, we yield this prime and the quotient. Otherwise, we just continue with the remainder of the prime list.

The result of this function, hence, is one prime factor and another number that, multiplied by the factor is n. If this other number is prime as well, we are done. Otherwise, we must continue factoring this other number:

```
trialfact :: Natural → [Natural]
trialfact 0 = []
trialfact 1 = []
trialfact n = let l = fromIntegral $ floor $ sqrt (fromIntegral n)
              in case findf n l allprimes of
                   (1, _) → [n]
                   (p, q) → if prime q then [p, q]
                                       else p : trialfact q
```

This function receives the number to be factored and yields the list of factors of this number. 0 and 1 cannot be factored. The result in this case, hence, is simply the empty list. For all other cases, we first determine the upper limit as the greatest natural number less than the square root of n. We already discussed the reasoning for this upper limit in the context of the Sundaram sieve: we assume an ordered list of primes and, when the current prime is greater than this limit, the square root of n, the product of any two primes greater than this prime will necessarily be greater than n. There is hence no need to continue the search.

We then call *findf* with n, the limit l and *allprimes*. If the result is $(1, \_)$, *i.e.* if *findf* has not found a proper solution, then n must be prime and we return $[n]$ as the only factor. For other results, we know that p, the first of the pair, is a prime. We do not know this for the second of the pair, which may or may not be prime. If it is prime, we yield $[p, q]$. Otherwise, we call *trialfact* with q and add p to the result.

We could skip the primality test and just continue with *trialfact* q, since, if q is prime, *findf* will yield $(1, q)$ and then *trialfact* would yield $[q]$ anyway. In the hope of finding a primality test that is more efficient then the test we have defined so far (which, itself, uses a sieve to construct *allprimes*), we use this explicit test to obtain some speed-up in the future.

We can use *trialfact* to investigate the distribution of primes further. We start with the numbers $2 \ldots 32$ and create the list of factorisations of these numbers calling
map trialfact $[2..32]$:

$[2]$
$[3]$
$[2, 2]$
$[5]$
$[2, 3]$
$[7]$
$[2, 2, 2]$
$[3, 3]$
$[2, 5]$
$[11]$
$[2, 2, 3]$
$[13]$
$[2, 7]$
$[3, 5]$
$[2, 2, 2, 2]$
$[17]$
$[2, 3, 3]$
$[19]$
$[2, 2, 5]$
$[3, 7]$
$[2, 11]$
$[23]$
$[2, 2, 2, 3]$
$[5, 5]$
$[2, 13]$
$[3, 3, 3]$
$[2, 2, 7]$
$[29]$
$[2, 3, 5]$
$[31]$
$[2, 2, 2, 2, 2]$

What jumps immediately into the eyes is the fact that the lists appear to grow in length – with sporadic prime numbers to appear in between slowing down the growth. Here is the list of the sizes of the factorisations $2 \ldots 32$:

1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 3, 1, 2, 2, 4, 1, 3, 1, 3, 2, 2, 1, 4, 2, 2, 3, 3, 1, 3, 1, 5.

Most factorisations $(2 \ldots 32)$ are of size 1 or 2, 1 being the size of prime number factorisations, which consists only of that prime number. Greater numbers appear sporadically, 3, 4 and 5, and seem to grow – in-line with our previous observation. Those greater numbers are certainly caused by repetition of prime numbers, such as $2 \times 2 \times 2 \times 2 = 16$. How would it look if we counted only unique primes? Let us have a try:

1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 2, 1, 2, 1, 3, 1, 1.

The factorisations, now, grow much slower. The first factorisation with more than 3 distinct primes appears only with 30 ($2 \times 3 \times 5$). All other factorisations have either size 1 or 2. Factorisations of size 1 are those of the primes and those containing only repeated primes. But there are no clear patterns that would reveal some regularity among factorisations. Perhaps, it could be helpful to look at the distinction odd versus even sized factorisations? To do that, we should distinguish between factorisations with and without repeated primes. We could, for instance, say that factorisations with repeated primes have the value 0; odd-sized factorisations have value -1 and even-sized factorisations have value 1.

This rule describes the Möbius function, which we could define as:

$$
\begin{array}{l}
\mathit{moebius} :: \mathit{Natural} \to \mathit{Integer} \\
\mathit{moebius} = \mathit{chk} \circ \mathit{trialfact} \\
\quad \textbf{where}\ \mathit{chk}\ f \mid f \not\equiv \mathit{nub}\ f \quad\ = 0 \\
\qquad\qquad\quad\ \mid \mathit{even}\ (\mathit{length}\ f) = 1 \\
\qquad\qquad\quad\ \mid \mathit{otherwise} \quad\ = -1
\end{array}
$$

The *moebius* function for a number n checks whether the factorisation of that number contains repeated primes ($f \not\equiv \mathit{nub}\ f$); if so, the result is 0. Otherwise, if the number of primes in the factorisation is even, the result is 1. Otherwise, the result is -1. Notice that we use *Integer* as output data type, since -1 is not a natural number.

Here are the values of the Möbius function for the numbers $2 \ldots 32$:

-1, -1, 0, -1, 1, -1, 0, 0, 1, -1, 0, -1, 1, 1, 0, -1, 0, -1, 0, 1, 1, -1, 0, 0, 1, 0, 0, -1, -1, -1, 0.

It is still difficult to see regularities. What, if we defined an accumulated Möbius function where each value corresponds to the sum of the values of the Möbius function up to the the current number:

$$
\begin{array}{l}
\mathit{mertens} :: \mathit{Natural} \to \mathit{Integer} \\
\mathit{mertens}\ n = \mathit{sum}\ (\mathit{map}\ \mathit{moebius}\ [1\,..\,n])
\end{array}
$$

Mapped on the numbers $2 \ldots 32$, this function gives:

0, -1, -1, -2, -1, -2, -2, -2, -1, -2, -2, -3, -2, -1, -1, -2, -2, -3, -3, -2, -1, -2, -2, -2, -1, -1, -1, -2, -3, -4, -4.

This still does not reveal convincing patterns. Apparently, most numbers have a negative value, but this is true only for the small section we are looking at. The result for *mertens* 39 and *mertens* 40, for instance, is 0. The values for 94 to 100 are all positive peaking with 2 at 95 and 96.

This investigation appears to remain fruitless and we should give it up at least for the moment. We will come back to Möbius and Mertens, however. Even if not visible on the surface, there is something about the concept.

The Möbius function was invented by August Ferdinand Möbius (1790 – 1868), a German mathematician and astronomer, student of Gauss in Göttingen. There are many unusual concepts discovered or developed by this man, for instance, the famous *Möbius strip*, a two-dimensional surface with the uncommon property of having only one side in three-dimensional space.

Franz Mertens (1840 – 1927), after whom the Mertens function is named, is less known. He proposed the Mertens function together with a conjecture concerning its growth that, if proven correct, could have been used to prove the *Riemann Hypothesis* on the distribution of primes. But, unfortunately, Meterns' conjecture was proven wrong and the Riemann Hypothesis remains an enigma until today.

1.5 Factoring Factorials

In the next sections, we will dive into group theory related to primes. There is a problem that makes a very nice link between factoring and the upcoming investigations: factoring factorials.

The factorial of a number n is defined as the product of all numbers $1 \dots n$: $1 \times 2 \times 3 \times \dots \times n$. The immediate consequence of this definition is that all primes in the range $1 \dots n$ are factors of $n!$ and that all factors of $n!$ are primes in the range $1 \dots n$. The first fact is easy to see: since we multiply all numbers $1 \dots n$ with each other, all primes in this range must be part of the product. Furthermore, all composites in this range are products of prime factors in this range and, hence, $n!$ is the product of products of the primes between 1 and n.

For the second fact to become clear, assume for a moment that there were a prime $p > n$ that is a prime factor of $n!$ That would mean that some product of primes $< n$ would result in that p. But that is impossible, since p is a prime. It is not a product of other primes and can therefore not result from multiplying other primes and can thus not be a prime factor of $n!$

To find the prime factors of $n!$ (and, hence, $n!$ itself), we have to ask how often each prime appears in the factorisation of $n!$? This leads to the question how many numbers in the range $1 \dots n$ are actually divisible by a given prime. This is easily answered, when we realise that the range $1 \dots n$ consists of n consecutive numbers. For any number a, every a^{th} number is divided by a. There are, hence, $\lfloor n/p \rfloor$ numbers that are divided by p. Let us look at the example $n = 6$ and $p = 2$. The product $n!$ consists of six numbers: $1, 2, 3, 4, 5, 6$. Every second number is even, namely 2, 4 and 6 itself. This is $6/2 = 3$ numbers. Therefore, 2 must appear at least 3 times as factor in $n!$

But wait: 2 appears 2 times in 4, since the factorisation of 4 is 2^2. How can we tell how many of the numbers in the range are divided by 2 more than once? Well, we just do the same, we divide 6 by 4, since every fourth number is divided by 4. Since $\lfloor 6/4 \rfloor = 1$,

there is only one number in the range $1 \ldots 6$ that is divided by 4 and, hence, divided twice by 2, *viz.* 4 itself. When we add the two results $\lfloor 6/2 \rfloor = 3$ and $\lfloor 6/4 \rfloor = 1$, we get 4. In other words, there are 3 numbers divided by 2 and 1 number divided by 2 twice. Therefore, 2 appears 4 times in the prime factorisation of 6! Let us check if this result is correct: $6! = 720$. The prime factorisation of 720 is *trialfact* 720: $[2, 2, 2, 2, 3, 3, 5]$. So the result is correct.

Let us try to confirm the result for the other primes ≤ 6, namely 3 and 5. $\lfloor 6/3 \rfloor$ is 2; there are hence two numbers divided by 3 and these numbers are 3 and 6. Since $3^2 = 9$ is already greater than 6, there is no number in the range of interest that is divided by 3 twice. Therefore, there are two occurrences of 3 in the prime factorisation of 6! $\lfloor 6/5 \rfloor$ is 1, which means there is only one number divided by 5, namely 5 itself. 5, therefore appears once in the factorisation of 6! With this approach, we arrive at the correct result: $6! = 2^4 \times 3^2 \times 5 = 720$.

We can implement this approach in Haskell to get a speed-up on the factorial computation compared to the laborious multiplication of all numbers $1 \ldots n$. We first implement the logic to find the number of occurrences for one prime:

```
pInFac :: Natural → Natural → Natural
pInFac n p = p ↑ (go p 1)
  where go q e = let t = n `div` q
                 in if t ⩽ 1 then t
                           else let e' = e + 1 in t + go (p ↑ e') e'
```

In *go*, which is called with a prime number p and and expononent $e = 1$, we first compute the quotient $\lfloor n/q \rfloor$. If this quotient is 1 or 0, we immediately yield this number. Otherwise, we continue with p raised to $e + 1$. That is, if $n = 6$ and $p = 2$, then we first compute $t = 6$ `` `div` `` 3, which is 2 and hence greater than 1. We now increment e, which, initially is 1, and call *go* with $2^2 = 4$ and $e = 1 + 1 = 2$.

In the next round t is 6 `` `div` `` 4, which is 1. We return immediately and add 1 to the previous value of t, which was 3, obtaining 4. The function overall yields p raised to the result of *go*, hence $2^4 = 16$.

We call this function in the following code:

```
facfac :: Natural → [Natural]
facfac n = go allprimes
  where go (p : ps) = let x = pInFac n p
    in if x ≡ 1 then [] else x : go ps
```

The function *facfac* results in a list of *Natural*, *i.e.* it yields the factors of $n!$, sucht that *product* (*facfac* n) is $n!$ It calls the internal function *go* on *allprimes*. On the first prime, it calls *pInFac* n p. If the result is 1, *i.e.* if we raised p to zero, we terminate with the empty list. Otherwise, we continue with the tail of *allprimes*.

Here are the results for the numbers $2 \ldots 12$:

$[2]$
$[2, 3]$
$[8, 3]$
$[8, 3, 5]$
$[16, 9, 5]$
$[16, 9, 5, 7]$
$[128, 9, 5, 7]$
$[128, 81, 5, 7]$
$[256, 81, 25, 7]$
$[256, 81, 25, 7, 11]$
$[1024, 243, 25, 7, 11]$

It would be very interesting, of course, to know how much faster *facfac* is compared to the ordinary recursive *fact*. Well, *fact* multiplies n numbers with each other. There are hence $n - 1$ multiplications. By contrast, *facfac* calls *pInFac* for every prime less than or equal to n. *pInFac* is somewhat more complex in computation than multiplication, but when the difference between n and the number of primes up to n is significant, then the difference between the cost of multiplication and that of *pInFac* does not matter. The question remains: how many primes are there among the first n numbers?

1.6 Arithmetic modulo a Prime

On the first sight, there is nothing special about arithmetic modulo a prime. It is plain modular arithmetic where the number to which all the number operations are taken modulo happens to be a prime. However, as we will see in this section, something very significant changes, when it actually is a prime. Let us first recall the properties of modular arithmetic and define a Haskell data type to model it.

As we have seen before, numbers modulo a number n repeat cyclicly, that is any number modulo n is a number in the range $0 \ldots n - 1$. When we think of the clock again, any number, independent of its size, reduces to a number $0 \ldots 11$ taken modulo 12: $1 \bmod 12$ is 1, $5 \bmod 12$ is 5, $10 \bmod 12$ is 10 and $13 \bmod 12$ is again 1, $17 \bmod 12$ is 5 and $22 \bmod 12$ is 10. In other words, the numbers modulo a number n form a finite groupoid (or magma) whith addition and multiplication, that is, addition and multiplication are closed under natural numbers modulo n:

$5 + 3 = 8 \bmod 12 = 8$
$5 + 10 = 15 \bmod 12 = 3$
$13 + 15 = 28 \bmod 12 = 4$
$15 + 17 = 32 \bmod 12 = 8$

and

$5 \times 3 = 15 \bmod 12 = 3$
$5 \times 10 = 50 \bmod 12 = 2$
$13 \times 15 = 195 \bmod 12 = 3$
$15 \times 17 = 255 \bmod 12 = 3$.

If we consider more complex sums and products of the form $a+b+\cdots+c$ and $a\times b\times\cdots\times c$, it becomes apparent that it is more efficient to take the terms and factors modulo n before applying the operation:

$13 \bmod 12 + 15 \bmod 12 = 1 + 3 = 4$
$15 \bmod 12 + 17 \bmod 12 = 3 + 5 = 8$

or:

$13 \bmod 12 \times 15 \bmod 12 = 1 \times 3 = 3$
$15 \bmod 12 \times 17 \bmod 12 = 3 \times 5 = 15 \bmod 12 = 3$.

Modular arithmetic looks a bit weired at the beginning, for instance $13+15 = 4$ definitely looks wrong. But, in fact, nothing special has changed. We see that the associativity law holds:

$13 + (15 + 17) \mod 12 = (13 + 15) + 17 \mod 12 = 9$,
$13 \times (15 \times 17) \mod 12 = (13 \times 15) \times 17 \mod 12 = 3$.

There is also an identity for each, addition and multiplication. For addition this is 0 (and, thus, all integer multiples of 12):

$0 \bmod 12 = 0$
$24 \bmod 12 = 0$
$36 \bmod 12 = 0$
...

For multiplication, the identity is 1 (and, hence, all multiples of 12 plus 1):

$1 \bmod 12 = 1$
$25 \bmod 12 = 1$
$37 \bmod 12 = 1$
...

It holds of course that for any a divisible by 12:

$$a + b \mod 12 = b$$

and

$$(a + 1) \times b \mod 12 = b.$$

The distributive law holds as well:

$$a \times (b + c) \mod n = ab + ac \mod n.$$

This altogether means that numbers modulo n form a semiring with addition and multiplication just as the natural numbers. The difference between natural numbers and numbers modulo n is that the set of natural numbers is infinite whereas the set $1 \ldots n-1$ is of course finite.

Let us have a look at how we can model such a modular semigroup with Haskell. We first define a data type *Module*:

```
data Module = Module Natural Natural
```

A *Module* according to this definition is created by two natural numbers. The first is the modulus n to which we take the second modulo. For instance, *Module* 12 13 is 13 mod 12. To enforce modular arithmetic from the beginning, we provide a constructor:

```
tomod :: Natural → Natural → Module
tomod n a = Module n (a `rem` n)
```

For all (binary) operations on *Module*s, we want to ensure that both parameters have the same modulus. Operating on two *Module*s with different moduli leads to wrong results. For this reason, we will use a guard on all operations:

```
withGuard :: (Module → Module → r) → Module → Module → r
withGuard o x@(Module n  a)
          y@(Module n' b) | n ≢ n'     = error "different moduli"
                          | otherwise = x `o` y
```

Now, we make *Module* instance of some type classes:

```
instance Show Module where
  show (Module n a) = show a
instance Eq Module where
  (≡) = withGuard (λ(Module _ a) (Module _ b) → a ≡ b)
instance Ord Module where
  compare = withGuard (λ(Module _ a) (Module _ b) → compare a b)
```

We *show* the number actually taken modulo n and consider n known in the context. This is much more convenient to read, even if some information is lost.

To check for equality and to compare two *Module*s we apply the guard to the operations, (≡) and *compare* respectively.

The next listing shows addition and multiplication:

$$
\begin{aligned}
&\mathit{add} :: \mathit{Module} \to \mathit{Module} \to \mathit{Module}\\
&\mathit{add}\ (\mathit{Module}\ n\ a)\ (\mathit{Module}\ \_\ b) = \mathit{Module}\ n\ ((a+b)\ `\mathit{rem}`\ n)\\
&\mathit{mul} :: \mathit{Module} \to \mathit{Module} \to \mathit{Module}\\
&\mathit{mul}\ (\mathit{Module}\ n\ a)\ (\mathit{Module}\ \_\ b) = \mathit{Module}\ n\ ((a*b)\ `\mathit{rem}`\ n)
\end{aligned}
$$

Since the numbers a and b are already modulo n, taking the results of the computations modulo n is an inexpensive operation. Since the maximum for both, a and b, is $n-1$, $a+b$ is at most $2n-2$ and, taking this modulo n, is just $2n-2-n = n-2$.

The greatest value the product $a \times b$ can achieve is $(n-1)(n-1) = n^2 - 2n + 1$. To reduce this value modulo n, one division step is needed and that is indeed the worst case for modular multiplication.

Now, what about subtraction? Subtracting two numbers modulo n should also be in the range $0 \ldots n-1$, but what happens, when the second number is greater than the first one? The normal subtraction beyond zero gives $a - b = -(b-a)$. In modular arithmetic, a negative number $-k$ is interpreted as $n-k$, *i.e.* the minus sign is interpreted as counting down from n, which in fact is the same as counting down from 0, since n mod n is just 0.

We can therefore, when we have a negative number in the range $-(n-1) \ldots 0$, just add n to the result: $a - b = -(b-a) + n = n - (b-a)$. For instance with $n = 12$: $3 - 9 + 12 = -6 + 12 = 6$. Note that subtraction handled like this is the inverse of addition: $3 - 9 = 6 \mod 12$ and $6 + 9 = 3 \mod 12$. The point is that addition modulo n with two numbers already modulo n is at most $2n-2$. The remainder of any number up to $2n-2$ is just this number minus n: $6+9 = 15-12 = 3$. For subtraction, a similar is true: the smallest value, subtraction can produce is $-(n-1)$ (in the case of $0 - 11$, for instance).

We, hence, can implement subtraction as:

$$
\begin{aligned}
&\mathit{sub} :: \mathit{Module} \to \mathit{Module} \to \mathit{Module}\\
&\mathit{sub}\ (\mathit{Module}\ n\ a)\ (\mathit{Module}\ \_\ b) \mid a < b \qquad = \mathit{Module}\ n\ (a + n - b)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{Module}\ n\ (a - b)
\end{aligned}
$$

Note that we change the order of the operations for the case that $a < b$. If we performed $a - b$ first, we would subtract beyond zero. Even if the overall result is again a natural number, the intermediate result is not. Therefore, we first add a and n and then we subtract b. In spite of handling problems of natural numbers only, we are on the verge of entering new territory. But we can state a very exciting result: In modular arithmetic, natural numbers and addition form a group: addition is closed, addition adheres to the associativity law, there is an identity, namely 0 (and all multiples of the modulus n), and, for any number modulo n, there is an inverse element.

With addition, subtraction and multiplication defined, we can now make *Module* an instance of *Num*:

instance *Num Module* **where**
  $(+)$ = *withGuard add*
  $(-)$ = *withGuard sub*
  $(*)$ = *withGuard mul*
  abs a = *a*
  signum (*Module n a*) = (*Module n* (*signum a*))
  fromInteger i = *Module* (*fromInteger* $(i+1)$) (*fromInteger i*)

The basic arithmetic operations are defined as *add*, *sub* and *mul* with the guard to avoid arithmetic on different moduli. As with natural numbers, we ignore *abs*, since all natural numbers are positive. *signum* is just the *signum* of a, *i.e.* a *Module* with 0 for 0 and 1 for any number greater than 0.

fromInteger is a bit tricky. We cannot convert an integer to a *Module* "as such". To convert an integer, we must know to which n the number should be taken modulo. When we say that, by default, an integer i is $i \bmod (i+1)$, the value of that module is always i, for instance: $2 \bmod 3 = 2$. This appears to be a reasonable default value.

Division, as usual, is not so easy. We would like to define division such that it serves as the inverse of multiplication, *e.g.* $a \times b/a = b$. That means that for any number a, we want a number a', such that $a \times a' = 1$. Consider the example $n = 6$ and $a = 3$:

$3 \times 0 = 0 \mod 6$
$3 \times 1 = 3 \mod 6$
$3 \times 2 = 0 \mod 6$
$3 \times 3 = 3 \mod 6$
$3 \times 4 = 0 \mod 6$
$3 \times 5 = 3 \mod 6.$

This looks strange: any multiplication of 3 modulo 6 creates either 0 or 3, but not 1,2,4 or 5. The point is that 3 divides 6, in particular $2 \times 3 = 6$. For this reason, 6 divides every second product of 3, *i.e.* $0, 6, 12, 18, 24, \ldots$ The other half of multiples are just those that leave a remainder of 3 divided by 6.

What, if a does not divide n, like for example with $a = 6$ and $n = 9$?

$6 \times 0 = 0 \mod 9$
$6 \times 1 = 6 \mod 9$
$6 \times 2 = 3 \mod 9$
$6 \times 3 = 0 \mod 9$
$6 \times 4 = 6 \mod 9$
$6 \times 5 = 3 \mod 9$
$6 \times 6 = 0 \mod 9$
$6 \times 7 = 6 \mod 9$
$6 \times 8 = 3 \mod 9.$

We see more variety, but, still, we have only 3 numbers out of 9 possible. Now, every

third multiple of a ($= 6$) is divisible by n ($= 9$). This is, of course, because 3 divides 9 and also divides 6. In consequence every third multiple of 6 is divisible by 9. When we think this through, we see that there are indeed many numbers n divisible by numbers $1 \ldots n-1$. Only if n is *coprime* to those numbers, all of them would appear as result of multiplication of any two numbers $a, b \in 1 \ldots n-1$. That two numbers, a and b, are coprime means that they have no common factors, *i.e.*: $\gcd(a, b) = 1$. If we look at an example, where n is coprime of all numbers $1 \ldots n-1$, we see that all numbers $0 \ldots n-1$ actually appear as results of the multiplications of one of them by all others in the range:

$3 \times 0 = 0 \mod 5$
$3 \times 1 = 3 \mod 5$
$3 \times 2 = 1 \mod 5$
$3 \times 3 = 4 \mod 5$
$3 \times 4 = 2 \mod 5.$

The point is that 5 has no common divisor with any of the numbers $0 \ldots 4$. We know that for sure, because 5 is a prime number. In consquence, no multiple of any number a from the range $1 \ldots 4$ will be divisible by 5 but those that are also multiples of 5. For this reason, any multiple of a number in the range that is not a multiple of 5 will again leave a remainder in the range when divided by 5. At the same time, the results of the multiplications of any two remainders must be different, that is, for any distinct $a, b, c, \in 1 \ldots 4$, if $ab = d$ and $ac = e$, then $d \neq e$. Otherwise, ab and ac would leave the same remainder with 5, which cannot be, since that would mean that there was a number k, such that $5k + ab = 5k + ac$, boiling down to $ab = ac$ and, by dividing a, $b = c$.

This is a significant result. It implies that, if we could devise an algorithm that finds the inverse of any a mod n (for n being prime), we would not only have a division algorithm, but we would have defined a multiplication group over natural numbers – and this is where arithmetic modulo a prime is different from arithmetic modulo a composite.

We find such an algorithm if we go to the heart of the matter. It is related to a special property of the gcd. We know that the Euclidean algorithm, which computes the gcd, proceeds by computing the remainder (which, for positive numbers, is the same as the modulo operation): $\gcd(a, b) = \gcd(b, a \bmod b)$. If c is the remainder, *i.e.* $c = a \bmod b$, then we have

$$c = a - qb, \tag{1.10}$$

where q is the quotient, *i.e.* the greatest number that multiplied with b is equal or less than a. This equation is equivalent to the following:

$$c = ka + lb, \tag{1.11}$$

where $k = 1$ and $l = -q$. We now prove by induction on gcd that, for any d such that $d = \gcd(a, b)$, there are two integers k and l, positive or negative, for which holds $d = ka + lb$. The base case is equation 1.11. We have to prove that, if 1.11 shows the n^{th} recursion step of the Euclidean algorithm gcd, then, in the $(n+1)^{th}$ rescursion step, it still holds for the remainder of the arguments in that iteration, d, that there are two integers m and n, such that $d = ma + nb$. Since the final result of gcd is the remainder of the previous recursion step, this proves that there are always two integers k and l such that $\gcd(a, b) = ka + la$.

If c in the equation $c = ka + lb$ represents the remainder in the n^{th} recursion step of *gcd*, then c is the second argument in the next recursion step and d in $d = b - qc$ represents the remainder in this step. We substitute the base case 1.11 for c:

$$d = b - qc = b - q(ka + lb) \tag{1.12}$$

Let us distinguish the quotient in equation 1.10 and the one in 1.12 by adding the subscripts: $c = a - q_1 b$ and $d = b - q_2 c$. Since $c = a - q_1 b$, we can state that $d = b - q_2(a - q_1 b)$ or, according to the base case 1.11:

$$d = b - q_2(ka + lb). \tag{1.13}$$

We multiply this out to get:

$$d = b - (q_2 ka + q_2 lb), \tag{1.14}$$

which is just

$$d = b - q_2 ka - q_2 lb. \tag{1.15}$$

By regrouping and adding $-q_2 lb + b$, we obviously get

$$d = -q_2 ka - (q_2 l - 1)b. \tag{1.16}$$

We set $m = -q_2 k$ and $n = -(q_2 l - 1)$ and obtain the desired result:

$$d = ma + nb. \quad \square \tag{1.17}$$

To illustrate this with an example, we claim that the remainder in the second iteration of $\gcd(21, 15)$ is $-21q_2 k - 15(q_2 l - 1)$, where k and l fulfil the equation $21k + 15l = (21 \bmod 15)$ and $l = -q_1$. So we have $21k - 15 = 6$, which becomes true if $k = 1$.

In the next round, we have $\gcd(15, 6)$. The quotient of 15 and 6, q_2, is 2. We, hence, claim that $21 \times -2 \times 1 - 15 \times (2 \times -1 - 1) = 15 \bmod 6 = 3$. Let us see if this is true. We simplify to $-42 - 15(-2 - 1)$, which in its turn is $-42 - 15(-3)$ or $-42 + 45 = 3$, which is indeed the expected result.

Now we will look at the special case that the gcd of two numbers a and b is 1. There still must be two numbers k and l, such that

$$1 = ka + lb. \tag{1.18}$$

From this, we can prove as a corollary, that if a prime p divides ab, it must divide either a or b, a fact that we took for granted, when we proved the fundamental theorem of arithmetic. If p does not divide a, then we have $\gcd(a, p) = 1$ and, hence, $1 = ka + lp$. Multiplying by b, we get $b = b(ka + lp)$ or $b = kab + lbp$. The fact that p divides ab means that there is a number r, such that $ab = rp$. So, we can also say $b = krp + lbp$ or $b = p(kr + lb)$, where p clearly appears as a factor of b. □

If we compute $\gcd(a, n)$, where n is a prime, we know we get 1 back and we know there must be two integers k and l such that $1 = ka + ln$. We can transform this equation by subtracting ln to $ka = 1 - ln$. Since ln is a multiple of n, $1 - ln$, which is the same as $-ln + 1$, would leave the remainder 1 on division by n, *i.e.* $-ln + 1 = 1 \mod n$. In other words: $ka = 1 \mod n$. That is actually what we are looking for: a number that, multiplied by a, is 1. k, hence, is the wanted inverse of a modulo n. The question now is: how to get to k?

There is a well known algorithm that produces not only the greatest common divisor, but also k and l. This algorithm is called the *extended greatest common divisor* or xGCD:

```
xgcd :: (Num a, Integral a) ⇒ a → a → (a, (a, a))
xgcd a b = go a b 1 0 0 1
  where go c 0 uc vc _ _  = (c, (uc, vc))
        go c d uc vc ud vd = let (q, r) = c `quotRem` d
                             in go d r ud vd (uc − q ∗ ud)
                                             (vc − q ∗ vd)
```

The listing, admittedly, looks somewhat confusing at the first sight. However, it bears the classic *gcd*. If you ignore the four additional parameters of *go*, you see that *go* calls *quotRem*, instead of just *rem* as *gcd* does, and it then recurses with *go d r*, *d* being initially b and r being the remainder – that is just *gcd*. But *go* additionally computes $uc - q \times ud$ and $vc - q \times vd$. We start with $uc = 1$, $vc = 0$, $ud = 0$ and $vd = 1$, hence: $1 - q \times 0 = 1$ and $0 - q \times 1 = -q$. These are just k and l after the first iteration. In the next iteration, we will have $uc = 0$, $vc = 1$, $ud = 1$ and $vd = -q_1$ and compute $0 - q_2 \times 1$ and $1 - q_1 \times -q_2$, which you will recognise as m and n from equation 1.16. The algorithm is just another formulation of the proof we have discussed above.

Let us look at *xgcd* with the example above, $a = 21$ and $b = 15$. We start to call *go* as

go 21 15 1 0 0 1, which is

$(1, 6) = 21$ `` `quotRem` `` 15

in

go 15 6 0 1 $(1 - 1 * 0)$ $(0 - q * 1)$
go 15 6 0 1 1 (-1).

This leads to

$(2, 3) = 15$ `` `quotRem` `` 6

in

go 6 3 1 (-1) $(0 - 2 * 1)$ $(1 - 2 * (-1))$
go 6 3 1 (-1) (-2) 3.

In the next round we have

$(2, 0) = 6$ `` `quotRem` `` 3

and we now call, ignoring ud and vd:

go 3 0 (-2) 3 _ _

and, since $d = 0$, just yield $(3, (-2, 3))$, where the k we are looking for is -2, *i.e.* the first of the inner tuple.

Since 15 and 21 in the example above are not coprime the remainder is not 1 but 3 (since $\gcd(21, 15) = 3$), When we use the function with a prime number p and any number $a < p$, the remainder is 1. The resulting k is the inverse of a mod p and we can therefore use this k to implement division. But, actually, we are not in Kansas anymore: k may be negative. That is, even if we are still discussing problems of natural numbers, we have to refer to negative numbers and, thus, use a number type we have not yet implemented.

Technically, this is quite simple – it hurts of course that we have to cheat in this way. Anyway, here is a simple solution:

$$
\begin{aligned}
&\mathit{nxgcd} :: \mathit{Natural} \to \mathit{Natural} \to (\mathit{Natural}, \mathit{Natural}) \\
&\mathit{nxgcd}\ a\ n = \mathbf{let}\ a' \qquad\quad = \mathit{fromIntegral}\ a \\
&\qquad\qquad\qquad\quad\ n' \qquad\quad = \mathit{fromIntegral}\ n \\
&\qquad\qquad\qquad\quad\ (r, (k, \_)) = \mathit{xgcd}\ a'\ n' \\
&\qquad\qquad\quad \mathbf{in\ if}\ k < 0\ \mathbf{then}\ (\mathit{fromIntegral}\ r, \mathit{fromIntegral}\ (k + n')) \\
&\qquad\qquad\qquad\qquad\ \mathbf{else}\ (\mathit{fromIntegral}\ r, \mathit{fromIntegral}\ k)
\end{aligned}
$$

This function is somewhat difficult to look through because of all the conversions. First we have to convert the natural numbers to integers using *fromIntegral*, then we have to convert the result back to a natural number, again, using *fromIntegral*. This works because both types, *Integer* and *Natural*, belong to class *Integral*.

After conversion, we apply *xgcd* on the integers ignoring l, just using the remainder and k. (The reason that we do not throw away the remainder as well is that we will need the remainder sometimes to check that $xgcd\ a\ b \equiv 1$.) Now, if k is a negative number, we add the modulus n to it, as we have learnt, when we studied subtraction.

Let us specialise *nxgcd* for the case that we only want to have the inverse:

$$\begin{aligned}&inverse :: Natural \to Natural \to Natural\\ &inverse\ a = snd \circ nxgcd\ a\end{aligned}$$

The inverses of the numbers modulo 5 are for instance:

$1 : inverse\ 1\ 5 = 1$
$2 : inverse\ 2\ 5 = 3$
$3 : inverse\ 3\ 5 = 2$
$4 : inverse\ 4\ 5 = 4.$

Note that there is no inverse for 0, since any number multiplied by 0 is just 0. Another way to state this is that $1/0$ is undefined.

Any other number a and its inverse a' behave as follows: $a \times a' = 1$ and, of course, $a \times b \times a' = b$. For instance:

$1 \times 1 = 1 \mod 5$
$2 \times 3 = 1 \mod 5$
$3 \times 2 = 1 \mod 5$
$4 \times 4 = 1 \mod 5.$

We can also play around like:

$2 \times 4 \times 3 = 4 \mod 5$
$3 \times 4 \times 4 = 3 \mod 5$
$3 \times 1001 \times 2 = 1001 = 1 \mod 5.$

Division, the inverse operation to multiplication, is now easily implemented as:

$$\begin{aligned}&mDiv :: Module \to Module \to Module\\ &mDiv\ (Module\ n\ a1)\ (Module\ \_\ a2) = Module\ n\ (((inverse\ a2\ n) * a1)\ `rem`\ n)\end{aligned}$$

With this function, we have a way to make *Module* member of the *Integral* class, but, before we can do that, we have to make it instance of the *Enum* and the *Real* classes, which is straight forward:

$$\begin{aligned}&\textbf{instance}\ Enum\ Module\ \textbf{where}\\ &\quad fromEnum\ (Module\ \_\ a) = fromIntegral\ a\\ &\quad toEnum\ i \qquad\qquad\quad = tomod\ (fromIntegral\ (i+1))\ (fromIntegral\ i)\\ &\textbf{instance}\ Real\ Module\ \textbf{where}\\ &\quad toRational\ (Module\ \_\ a) = fromIntegral\ a\end{aligned}$$

The *Integral* instance is now simply defined as:

```
instance Integral Module where
  quotRem x@(Module n _) y = (withGuard mDiv x y, Module n 0)
  toInteger (Module _ a)   = fromIntegral a
```

For *quotRem*, *mDiv* is used to compute the quotient and, since we have defined division in terms of multiplication, we know that there is never to be a remainder different from 0. We, hence, just return a *Module* with the value 0 as remainder of *quotRem*.

We could now go even further and define an instance for *Fractional*:

```
instance Fractional Module where
  (/) = mDiv
  fromRational = ⊥
```

It is nice to have the division operator available for modules, so we can do things like $a \,/\, b$, where a and b are of type *Module*. For *fromRational*, however, which is mandatory for defining the *Fractional* class, we have, for the time being, no good implementation.

There is an important corollary that follows from the invertibility of numbers modulo a prime, namely that any number in the range $1 \ldots p-1$ can be created by multiplication of other numbers in this range and for any two numbers a and n, there is unique number b that fulfils the equation

$$ax = n. \tag{1.19}$$

In other words: There are no primes modular a prime. For natural numbers with ordinary arithmetic, this is clearly not true. There is for instance no solution for equations like $3x = 2$ or $3x = 5$. In arithmetic modulo a prime, however, you always find a solution, for instance: $3x = 2 \mod 5$ has the solution 4, since $3 \times 4 = 12 = 2 \mod 5$.

This follows immediately from invertibility, since we only have to multiply n to the inverse of a to find x. If we have the inverse a' of a, such that

$$aa' = 1, \tag{1.20}$$

we just multiply n on both sides and get:

$$naa' = 1n. \tag{1.21}$$

For the example above, $3x = 2 \mod 5$, we can infer x from

$$3 \times 2 = 1 \mod 5 \tag{1.22}$$

by multiplying 2 on both sides:

$$2 \times 3 \times 2 = 2 \mod 5. \tag{1.23}$$

$ax = 2 \mod 5$, hence, has the solution $x = 4$. Here is a kind of magic square for numbers modulo 5:

	1	2	3	4
1	1	3	2	4
2	2	1	4	3
3	3	4	1	2
4	4	2	3	1

The leftmost column shows a multiplication result. The multiplication is defined as: $row_1 \times row_n$. The second row, with 1 in the first column, shows the inverse for each number: The inverse of 1 is 1; the inverse of 2 is 3; the inverse of 3 is 2 and the inverse of 4 is 4. The next row shows the multiplications resulting in 2: 1×2, 2×1, 3×4 and 4×3.

Let us summarise what we have learnt. Arithmetic modulo a prime p constitutes a finite field of the numbers $0 \ldots p-1$. The arithmetic operations on numbers modulo p always yield a number in that range, *i.e.* the operations are closed modulo p. Additionally to the properties we had already seen for natural numbers, associativity, identity and commutativity, we saw that operations modulo p are invertible for both addition and multiplication. In spite of the observation that all numbers we are dealing with are positive integers, *i.e.* natural numbers, subtraction and division are closed and every number modulo a prime has an inverse number for addition and multiplication. For the multiplicative group of the field, 0 must be excluded, since there is no number k such that $0 \times k = 1$ or, stated differently, $1/0$ is undefined. This, however, is true for all multiplicative groups.

Before going on, let us look at the gcd and the *xgcd* once more. It would be interesting to have a function that finds the GCD not only for two numbers, but for a list of numbers. For instance, the GCD of the numbers $6, 9, 12$ is 3, while that of $6, 9, 11$ is 1. The implementation for the gcd algorithm is in fact quite simple. We, obviously, have

$$\gcd(a, b, c) = \gcd(a, \gcd(b, c)) \tag{1.24}$$

and can therefore just fold the list with gcd:

```
mgcd :: (Integral a) ⇒ [a] → a
mgcd [] = 1
mgcd (i : is) = foldl' gcd i is
```

Note that we define the case with the empty list as 1. That is just conventional; one could also leave it undefined. But then users would have to check explicitly for this case.

The extended gcd algorithms is a not so simple. The issue is the integers which we called k and l above. Consider a list of the form $[a, b, c]$. We can compute the GCD as $\gcd(a, \gcd(b, c))$; but now we have

$$\gcd(a, \gcd(b, c)) = k_1 a + k_2 \gcd(b, c) \tag{1.25}$$

for two integers k_1 and k_2. The formula for computing these integers continues recursively into the second gcd, *i.e.*:

$$\gcd(b, c) = k_3 b + k_4 c \tag{1.26}$$

Since we multiply $\gcd(b, c)$ by k_2 in 1.25 above, we also need to multiply the integers within the second gcd by k_2, so that we finally get

$$\gcd(a, b, c) = k_1 a + k_2 k_3 b + k_2 k_4 c. \tag{1.27}$$

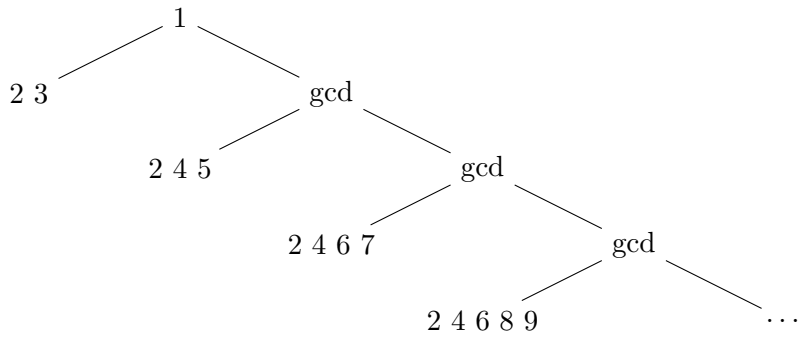
With longer lists the schema continues into the following gcds. With one more element in the list we would get

$$\gcd(a, b, c, d) = k_1 a + k_2 k_3 b + k_2 k_4 k_5 c + k_2 k_4 k_6 d. \tag{1.28}$$

This leads to the following pattern:

1
2 3
2 4 5
2 4 6 7
2 4 6 8 9
2 4 6 8 10 11
...

which corresponds to a binary tree that always branches at the right kid like this:



If we had the ks in one list, we could generate this structure with a simple fold-like function that we call *distr* for "distribute right":

$$
\begin{aligned}
&\mathit{distr} :: (a \to a \to a) \to a \to [a] \to [a] \\
&\mathit{distr}\ f\ n\ [\,] = [\,] \\
&\mathit{distr}\ f\ n\ xs = \mathit{go}\ n\ xs \\
&\quad \textbf{where}\ \mathit{go}\ p\ [\,] = [p] \\
&\qquad\qquad \mathit{go}\ p\ [k] = [f\ p\ k] \\
&\qquad\qquad \mathit{go}\ p\ (k : ks) = \textbf{let}\ h = \mathit{head}\ ks \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (f\ p\ k) : \mathit{go}\ (f\ p\ h)\ (\mathit{tail}\ ks)
\end{aligned}
$$

The function has a snag for lists with an odd number of elements greater than 3. For 3 elements, say $[1, 2, 3]$, the function called as *distr* $(*)$ 1 $[1, 2, 3]$ would result in

$[1, 2 \times 3]$.

That's fine. On four elements (*e.g.* $[1, 2, 3, 4]$), it would correctly generate

$[1, 2 \times 3, 2 \times 4]$.

But for five elements, it would generate

$[1, 2 \times 3, 2 \times 4 \times 5]$.

What we would like to have is, however,

$[1, 2 \times 3, 2 \times 4, 2 \times 5]$,

since the last two elements are both leaves. Anyway, we accept this shortcoming in favour of generality of the *distr* function. As we will see, we simply won't call *distr* with the critical case.

Here is a first approach to implementing the *mxgcd*:

$$
\begin{aligned}
&\mathit{mxgcd2} :: (\mathit{Integral}\ a) \Rightarrow [a] \to (a, [a]) \\
&\mathit{mxgcd2}\ [\,] \ = (1, [\,]) \\
&\mathit{mxgcd2}\ [x] = (x, [1]) \\
&\mathit{mxgcd2}\ \mathit{as} \ = \mathbf{let}\ (g, \mathit{rs}) = \mathit{go}\ \mathit{as}\ \mathbf{in}\ (g, \mathit{ks}\ \mathit{rs}) \\
&\quad \mathbf{where}\ \mathit{go}\ [i, j] \ \ = \mathbf{let}\ (g, (x, y)) = \mathit{xgcd}\ i\ j \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (g, [x, y]) \\
&\qquad\qquad \mathit{go}\ (i : \mathit{is}) = \mathbf{let}\ (\mathit{g0}, \mathit{rs}) = \mathit{go}\ \mathit{is} \\
&\qquad\qquad\qquad\qquad\qquad\ \ (g, (x, y)) = \mathit{xgcd}\ i\ \mathit{g0} \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (g, [x, y] +\!\!+ \mathit{rs}) \\
&\qquad\qquad \mathit{ks} = \mathit{distr}\ (*)\ 1
\end{aligned}
$$

The function creates a tuple of the form $(a, [a])$, where the first represents the GCD and the second represents the list of ks. If we call *mxgcd* like this

$(g, ks) = \mathit{mxgcd2}\ \mathit{xs}$

the following constraint holds:

$g \equiv \mathit{sum}\ [k * x \mid (k, x) \leftarrow \mathit{zip}\ \mathit{ks}\ \mathit{xs}$

The function first handles some trivial cases, namely the empty list and a list consisting of only one element. It then calls *go* with the argument passed in.

There are two cases for *go*, namely a list with two elements and a list with more than two elements. (Note that *go* is never called with only one or no element in the list.)

For a list with two elements, the *xgcd* is called with those two elements. The result is just the result of *xgcd*, but with the integers x and y as elements of a list, not a tuple, to match the type signature of the function.

For a list with more than two elements, we first recursively call *go* on the tail of the list from which we obtain the GCD of the tail (*g0*) and a list of integers. (Note that the tail of the list, *is*, always has at least two elements, otherwise we would have entered the first case. This way it is guaranteed that we will never create an unhandled pattern.) We then compute the *xgcd* on the head and the intermediate *g0*. The result is the new GCD and the list of integers with the new integers added to it. Finally, we apply *distr* $(*)$ 1 on this list.

The result list *go* is guaranteed to contain an even number of integers because each gcd generates two integers. The number of elements in the result list is therefore $2g$ where g is the number of gcd calls and, hence, always even; g is $n - 1$, for n the number of elements in the input list. The number of elements in the intermediate result list is thus $2(n-1)$. It is therefore safe to use *distr*: the cases with an odd number of elements will not arise from this usage.

But the function is not perfect. In particular, it is not tail-recursive, since most of the work is done after the recursive call to *go* leading to a deep stack that must be unwound afterwards. This is a result of the structure we implemented, namely to "fold" to the

series of calls of the form:

$$\gcd(a, \gcd(b, \gcd(c, \ldots)))$$

We could do the opposite, *i.e.*, compute the $\gcd(a, b)$ and go into the recursion with the result. We would then get a structure like the following:

$$\gcd(\ldots, (\gcd(\gcd(a, b), c))$$

Here is an implementation:

$$\begin{aligned}
&\mathit{mxgcd} :: [\mathit{Integer}] \to (\mathit{Integer}, [\mathit{Integer}]) \\
&\mathit{mxgcd}\ [\,] = (1, [\,]) \\
&\mathit{mxgcd}\ [x] = (x, [1]) \\
&\mathit{mxgcd}\ (a : \mathit{as}) = \mathbf{let}\ (g, \mathit{rs}) = \mathit{go}\ [\,]\ a\ \mathit{as}\ \mathbf{in}\ (g, \mathit{reverse}\ (\mathit{ks}\ \mathit{rs})) \\
&\quad \mathbf{where}\ \mathit{go}\ \mathit{rs}\ i\ [j] = \mathbf{let}\ (g, (x, y)) = \mathit{xgcd}\ i\ j \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ (g, [y, x] +\!\!+ \mathit{rs}) \\
&\qquad\qquad \mathit{go}\ \mathit{rs}\ i\ \mathit{is}\ \ = \mathbf{let}\ (g, (x, y)) = \mathit{xgcd}\ i\ (\mathit{head}\ \mathit{is}) \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \mathit{go}\ ([y, x] +\!\!+ \mathit{rs})\ g\ (\mathit{tail}\ \mathit{is}) \\
&\qquad\qquad \mathit{ks} = \mathit{distr}\ (*)\ 1
\end{aligned}$$

In this variant, we first compute the *xgcd* of the first pair and advance only then into the next recursion. Two difficulties arise: we need a second element to which we can apply the first call of *xgcd* and, second, we need to remember the result list to which to add the new result when we return from the recursion.

To solve the first issue, we split the list into head and tail and apply the first *xgcd* on the head and the head of the tail. We then pass the result of the *xgcd* along, *viz.* the GCD. The next *xgcd* will then be computed with this GCD and the next element in the list.

We further pass along the result list (which initially was the empty list) with the two integers obtained from *xgcd* added to it. Note that these elements are added in reverse order. We do this because we construct the gcd calls in reverse order compared to the *mxgcd2* implementation and must, hence, apply the multiplications in reverse order too. Accordingly, the last step of the algorithm is to reverse the result of *distr* $(*)$ 1.

1.7 Congruence

There is an important fact that, in the light of modular arithmetic, appears to be completely trivial, namely that all numbers $0 \ldots n-1$ leave the same remainder divided by n as infinitely many other numbers $\geq n$. For instance, 0 leaves the same remainder as n divided by n; 1 leaves the same remainder as $n+1$; 2 leaves the same remainder as

$n+2$ and so on. Furthermore, 1 leaves the same remainder as $2n+1$, $3n+1$, $4n+1$, ... This relation, that two numbers leave the same remainder divided by another number n, is called congruence and is written:

$$a \equiv b \pmod{n}.$$

We have for example:

$$1 \equiv mn + 1 \pmod{n} \tag{1.29}$$

$$2 \equiv mn + 2 \pmod{n} \tag{1.30}$$

$$k \equiv mn + k \pmod{n} \tag{1.31}$$

An important congruence system is Fermat's *Little Theorem*, which is called like this to distinguish it from the other famous theorem by Pierre Fermat, his *Last Theorem*, which, in its turn, is named this way, because, for many centuries, it was the last of Fermat's propositions that was not yet proven.

Fermat's little theorem states that, for any integer a and any prime number p:

$$a^p \equiv a \pmod{p}, \tag{1.32}$$

which is the same as:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{1.33}$$

That the two equations are equivalent is seen immediately, when we multiply both sides of the second equation with a: $a \times a^{p-1} = a^p \equiv a \times 1 = a \pmod{p}$.

One of the proofs of the little theorem brings two major themes together that we have already discussed, namely binomial coefficients and modular arithmetic. You may have observed already that all binomial coefficients $\binom{p}{k}$, where p is prime and $0 < k < p$, are multiples of p. For instance:

$\binom{3}{2} = 3$
$\binom{5}{2} = 10$
$\binom{5}{3} = 10$
$\binom{7}{2} = 21$
$\binom{7}{3} = 35$
$\binom{7}{4} = 35$
$\binom{7}{5} = 21.$

This is not true for coefficients where p is not prime. For instance:

$\binom{4}{2} = 6$
$\binom{6}{2} = 15$
$\binom{8}{2} = 28$
$\binom{8}{4} = 70.$

To prove this, we first observe that all binomial coefficients are integers. Since we hardly know anything but natural numbers, we will not prove this fact here, but postpone the discussion to the next chapter. One way to define binomial coefficients is by means of factorials:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{1.34}$$

We transform this equation multiplying $k!(n-k)!$ on both sides:

$$n! = \binom{n}{k} k!(n-k)!, \tag{1.35}$$

which shows that n must divide either $\binom{n}{k}$ or $k!(n-k)!$, because the product of these two factors equals $n!$, which is a multiple of n by definition. Note that this is just the application of Euclid's lemma to a prime number. Since a prime has no factors to share with other numbers but itself, it must divide at least one of the factors of a product that it divides.

Let us check if n divides $k!(n-k)!$ Again, to divide the whole, n must divide one of the factors, either $k!$ or $(n-k)!$ But, if n is prime and $0 < k < n$ and $0 < n-k < n$, it cannot divide either of them, since none of the factors of $k!$ $(1 \times 2 \times \cdots \times k)$ and none of the factors of $(n-k)!$ $(1 \times 2 \times \cdots \times (n-k))$ is divided by n or divides n. One cannot compose a number that is divisible by a prime by multiplying only numbers that are smaller than that prime. We could do so easily for composites. $4! = 24$, for instance, is divisible by 8. But no number smaller than a given prime multiplied by another number smaller than that prime, will ever be divided by that prime. So, obviously, n must divide $\binom{n}{k}$ or, in other words, $\binom{n}{k}$ is a multiple of n. □

To the delight of every newcomer, it follows immediately from this fact that, if p is prime:

$$(a+b)^p \equiv a^p + b^p \pmod{p}. \tag{1.36}$$

This identity, for understandable reasons, is sometimes called *Freshman's Dream*. The binomial theorem, which we have already proven, states:

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

Since all $\binom{n}{k}$ for $0 < k < n$ are multiples of n if n is prime, they are all $0 \bmod n$. (Remember that, in modular arithmetic, we can take the modulo at any point when calculating a complex formula!) In the summation, hence, all terms for the steps $0 < k < n$ are 0 and only the first case, $k = 0$, and the last case, $k = n$, remain, whose coefficients are $\binom{n}{0} = 1$ and $\binom{n}{n} = 1$. The resulting formula, hence, is

$$(a+b)^n = \binom{n}{n}a^n + \binom{n}{0}b^{n-0} = a^n + b^n \quad \square$$

We will now prove Fermat's little theorem by induction. We choose the base case $a = 1$. Since $1^p = 1$, it trivially holds that $1^p \equiv 1 \pmod{p}$. We now have to prove that, if $a^p \equiv a \pmod{p}$ holds, it also holds that

$$(a+1)^p \equiv a + 1 \pmod{p}. \tag{1.38}$$

$(a+1)^p$ is a binomial formula with a prime exponent. We have already shown that $(a+b)^p = a^p + b^p \mod p$ and we can therefore conlcude $(a+1)^p = a^p + 1^p \mod p$, which, of course, is just $a^p + 1$. From the base case we know that $a^p \equiv a \pmod{p}$ and can therefore further conclude that $a^p + 1 \equiv a + 1 \pmod{p}$. $\square$

Another interesting congruence system with tremendous importance in cryptography is the *Chinese Remainder Theorem*. The funny name results from the fact that systems related to this theorem were first investigated by Chinese mathematicians, namely Sun Tzu, who lived between the 3^{rd} and the 5^{th} century, and Qin Jiushao, who provided a complete solution in his "Mathematical Treatise in Nine Sections" published in the mid-13^{th} century.

The theorem deals with problems of congruence systems where the task is to find a number x that leaves given remainders with given numbers. We can state such systems in general as follows:

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\dots \\ x &\equiv a_r \pmod{n_r} \end{aligned}$$

The theorem now states for x, $a_1 \dots a_r$ and $n_1 \dots n_r \in \mathbb{N}$ that, if the numbers $n_1 \dots n_r$ are coprime, then there is always a solution for x, which even further is unique modulo $\prod n_i$, the product of all the ns and, since the ns are coprime to each other, their least common multiplier.

Before we prove this theorem, let us look at potential algorithms to solve such systems. The first account would be "common sense": we would just search "brute-force" for the

proper solutions among candidates. Candidates are all numbers congruent to $a_1 \ldots a_r$ modulo $n_1 \ldots n_r$. For each pair of (a_i, n_i), we would create a list of congruences. Solutions would be the numbers that are in all such lists, *i.e.* the intersection of those lists. We would start by creating lists of congruences. The first element in the list of congruences for a pair (a_i, n_i) would be a_i, the next would be $a_i + n_i$ (since that number leaves the same remainder as a_i divided by n_i):

$$\begin{aligned}&\mathit{congruences} :: \mathit{Natural} \to \mathit{Natural} \to [\mathit{Natural}]\\&\mathit{congruences}\ a\ n = a : \mathit{congruences}\ (a + n)\ n\end{aligned}$$

This function will create an infinite list of all numbers leaving the same remainder with n (which is n_i) as a (which is a_i). For $a = 2$ and $n = 3$, *i.e.* the congruence $x \equiv 2 \pmod 3$, we would generate the list:

$[2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, ..]$

We would then devise a function to apply this generator to all pairs of (a_i, n_i) in the congruence system at hand:

$$\begin{aligned}&\mathit{mapCongruences} :: \mathit{Int} \to [\mathit{Natural}] \to [\mathit{Natural}] \to [[\mathit{Natural}]]\\&\mathit{mapCongruences}\ l\ as\ ns = [\mathit{take}\ l\ (\mathit{congruences}\ a\ n) \mid (a, n) \leftarrow \mathit{zip}\ as\ ns]\end{aligned}$$

This function receives three arguments: l of type *Int* (which should rather be *Natural*, but *Int* is chosen for convenience, since it is used with *take*) and *as* and *ns* both of type $[\mathit{Natural}]$. The function simply applies all pairs of (a, n) to the *congruences* generator. It limits the length of resulting lists from the generator to l. Otherwise, the list comprehension would never come to a result that we could then use to find the solution. Calling *mapCongruences* for the simple congruence system

$$\begin{aligned}x &\equiv 2 \pmod 3\\x &\equiv 3 \pmod 4\\x &\equiv 1 \pmod 5\end{aligned}$$

with $l = 10$ yields three lists:

$[2, 5, 8, 11, 14, 17, 20, 23, 26, 29]$
$[3, 7, 11, 15, 19, 23, 27, 31, 35, 39]$
$[1, 6, 11, 16, 21, 26, 31, 36, 41, 46]$.

Now we just have to intersect these lists:

$$\begin{aligned}&\mathit{chinese1} :: [\mathit{Natural}] \to [\mathit{Natural}] \to [\mathit{Natural}]\\&\mathit{chinese1}\ as\ ns = \mathbf{case}\ \mathit{mapCongruences}\ (\mathit{fromIntegral}\ \$\ \mathit{product}\ ns)\ as\ ns\ \mathbf{of}\\&\qquad\qquad [\,] \to [\,]\\&\qquad\qquad cs \to \mathit{foldr}\ \mathit{intersect}\ (\mathit{head}\ cs)\ (\mathit{tail}\ cs)\end{aligned}$$

This function receives two arguments, the list of $a_1 \dots a_r$ and the list of $n_1 \dots n_r$. On these lists, it calls *mapCongruences* with $l = \prod n_i$. The idea behind this choice will become clear in a minute. If the result of this application is an empty list, we return an empty list. This is just a trick to avoid an exception in the case where the input consists of empty lists. Otherwise, we fold the result list with *intersect* and *head cs* as the base case for *foldr*. The result for *chinese1* $[2, 3, 1]$ $[3, 4, 5]$ is

$[11, 71, 131]$.

If the theorem is correct, the three numbers in the resulting list should be congruent to each other modulo $3 \times 4 \times 5 = 60$. The call *map* (‘*rem*‘60) $[11, 71, 131]$, indeed, yields $[11, 11, 11]$.

The fact that the solution is unique modulo the product of all ns implies that there must be at least one solution in the range $a_s \cdots \prod n_i$, where a_s is the smallest of the as in the system. There is not necessarily a solution in the range of any particular n. But if there was no solution less than the product $\prod n_i$, there trivially would be no solution at all.

To guarantee that we look at all candidates up to $\prod n_i$, we take lists of the length $\prod n_i$. This is certainly exaggerated, since in lists with that number of elements, there are already much greater numbers. But it guarantees that we will find a solution.

This brute-force algorithm is quite instructive, since it shows the structure of the problem quite well. On the other hand, it is inefficient in terms of computational complexity. For big systems and, in particular, for large ns the congruence lists become unmanageably large. We should look for an algorithm that exploits our knowledge on modular arithmetic.

To start with, we observe that, obviously, all ns divide $\prod n_i$ (which we will call pN in the remainder of this section). But, since the ns are coprime to each other, n_i would not divide the product of all numbers but itself, that is $\frac{pN}{n_i}$. In other words $gcd(\frac{pN}{n_i}, n_i) = 1$. There, hence, exist two integers, k and l, such that $k \times \frac{pN}{n_i} + ln_i = 1$. Since ln_i is a multiplie of n_i, this means that $k \times \frac{pN}{n_i} = 1 \bmod n_i$. The number k is thus the inverse of $\frac{pN}{n_i} \bmod n_i$. Let us call the product of k and $\frac{pN}{n_i}$ (of which we know that it is 1 mod n_i) e_i: $e_i = k \times \frac{pN}{n_i}$.

We could now write the ridiculous formula $x \equiv e_i \times a_i \pmod{n_i}$ where we multiply e_i with a_i in the corresponding line of the congruence system. The formula is ridiculous, because we already know that $e_i \bmod n_i = 1$, the formula, hence, says $x \equiv 1 \times a_i \pmod{n_i}$, which adds very little to the original formulation $x \equiv a_i \pmod{n_i}$. But, actually, this stupid formula leads directly to the solution.

Note that for any $n_j, j \neq i$, n_j divides $\frac{pN}{n_i}$ and all its multiples including $e_i = k \times \frac{pN}{n_i}$. That is, for any e_i: $e_i \equiv 0 \pmod{n_j}, j \neq i$. So we could create the following, equally ridiculous equation:

$$x \equiv \sum_{j=1}^{r} e_j \times a_i \pmod{n_i}. \tag{1.39}$$

Since, for any specific i, all $e_j, j \neq i$, are actually 0 and for the one case, where $j = i$, e_i is 1, this equation is trivially true for any line in the system. It just states $x \equiv a_i \pmod{n_i}$. In other words: this sum fits all the single lines of the congruence system.

This trivially magic sum not taken modulo to any of the individual ns is of course a number that is much larger – or, much smaller, *i.e.* a large negative number – than the smallest number that would fulfil all congruences in the system. The unique solution, however, is this number taken modulo pN. Since pN is just a multiple of any of the n_i in the system, all numbers leaving the same remainder modulo pN will leave the same remainder modulo a specific n_i.

This approach to Chinese remainder systems is much more efficient than the brute-force logic we implemented before. To implement it in Haskell, we first implement the function that finds e_i, using the extended *gcd*:

```
inv :: Integer → Integer → Integer
inv n pN = let b           = pN `div` n
               (_, (k, _)) = xgcd b n
           in  k * b
```

Then we call this function for each pair (a_i, n_i) in the system, sum the products $a_i \times e_i$ and yield the result modulo pN:

```
chinese :: [Integer] → [Integer] → Natural
chinese as ns = let pN = product ns
                    es = [inv n pN | n ← ns]
                    e  = sum [a * e | (a, e) ← zip as es]
                in  fromIntegral (e `nmod` pN)
```

Since we are working with integers here instead of natural numbers – giving up to pretend that we can solve all problems related to natural numbers with natural numbers alone – we use a *mod* operator that is modelled on the *Module* data type defined in the previous section:

```
nmod :: Integer → Integer → Integer
nmod x n | x < 0     = n - ((-x) `rem` n)
         | otherwise = x `rem` n
```

Consider the example we already used above:

$$
\begin{aligned}
x &\equiv 2 \pmod 3 \\
x &\equiv 3 \pmod 4 \\
x &\equiv 1 \pmod 5.
\end{aligned}
$$

We would solve this system by calling *chinese* $[2,3,1]$ $[3,4,5]$. The results for *inv* are:

$inv\ 3\ 60 = -20$
$inv\ 4\ 60 = -15$
$inv\ 5\ 60 = -24.$

We would now call:

$sum\ [2 * (-20), 3 * (-15), 1 * (-24)] = sum\ [-40, -45, -24] = -109$

and take the result modulo 60: $(-109)\ `nmod`\ 60 = 11$. Confirm that this result fulfils the system:

$$
\begin{aligned}
11 &\equiv 2 \pmod 3 \\
11 &\equiv 3 \pmod 4 \\
11 &\equiv 1 \pmod 5.
\end{aligned}
$$

1.8 Quadratic Residues

Quadratic residues of a number n are natural numbers congruent to a perfect square modulo n. In general, q is a quadratic residue modulo n if:

$$x^2 \equiv q \pmod n. \tag{1.40}$$

A simple function to test whether a number q is indeed a quadratic residue with respect to another number x could look like this:

```
isResidue :: Natural → Natural → Natural → Bool
isResidue q n x = (x ↑ 2) `rem` n ≡ q
```

This function is a nice test, but it does not help us to find residues. The following function does that:

```
residues :: Natural → [Natural]
residues n = sort (nub [(x ↑ 2) `rem` n | x ← [0 .. n − 1]])
```

residues finds all residues modulo n by simply taking the remainder of the squares of all numbers $0 \ldots n-1$. Note that these are all remainders of squares modulo this number.

Any other square, for instance the square $(n+1)^2$, will reduce to one of the remainders in the range $0 \ldots n-1$. $(n+1)^2$ would just reduce to the remainder 1; $(n+2)^2$ would reduce to the remainder 2; likewise $(mn+1)^2$ would reduce to 1 or, in general, any number of the form $(mn+r)^2$, where r is a number from the range $0 \ldots n-1$, will always reduce to r.

Since *residues* tests all numbers in the range $0 \ldots n-1$, some numbers may appear more than once. The residues of 6, for instance, are: $0, 1, 4, 3, 4, 1$, since

$$\begin{aligned} 0^2 = 0 &\equiv 0 \pmod 6 \\ 1^2 = 1 &\equiv 1 \pmod 6 \\ 2^2 = 4 &\equiv 4 \pmod 6 \\ 3^2 = 9 &\equiv 3 \pmod 6 \\ 4^2 = 16 &\equiv 4 \pmod 6 \\ 5^2 = 25 &\equiv 1 \pmod 6 \end{aligned}$$

The function, therefore, *nub*s the result and sorts it for convenience.

Let us look at the residues of some small numbers:

$\mathit{residues}\ 9 = [0, 1, 4, 7]$
$\mathit{residues}\ 15 = [0, 1, 4, 6, 9, 10]$
$\mathit{residues}\ 21 = [0, 1, 4, 7, 9, 15, 16, 18]$

What happens, when the modulus is prime? Some examples:

$\mathit{residues}\ 3 = [0, 1]$
$\mathit{residues}\ 5 = [0, 1, 4]$
$\mathit{residues}\ 7 = [0, 1, 2, 4]$
$\mathit{residues}\ 11 = [0, 1, 3, 4, 5, 9]$
$\mathit{residues}\ 13 = [0, 1, 3, 4, 9, 10, 12]$

Apparently, the number of residues per modulus is constantly growing. In the case of prime moduli, however, there appears to be a strict relation between the modulus and the number of residues. There seem to be roughly $p/2$ residues for a prime modulus p or, more precisely, there are $(p+1)/2$ residues (if we include 0). This is not the fact with composite moduli. Let us devise a function that may help us to further investigate this fact:

$$\begin{aligned} &\mathit{countResidues} :: \mathit{Natural} \to \mathit{Int} \\ &\mathit{countResidues} = \mathit{length} \circ \mathit{residues} \end{aligned}$$

Applied to a random sequence of composite numbers the result appears to be random too (besides the fact that the number of residues is slowly growing together with the moduli):

9	15	21	25	26	30	32	35	90	100	150	500
4	6	8	11	14	12	7	12	24	22	44	106

Applied on prime numbers the result is always $(p+1)/2$:

3	5	7	11	13	17	19	23	29	31	37	41
2	3	4	6	7	9	10	12	15	16	19	21

and so on. There is, however, one remarkable exception, namely 2: *residues* $2 = [0, 1]$ and, hence, *countResidues* $2 = 2$. The general rule is therefore that, for an **odd** prime p, there are $(p+1)/2$ residues and $(p-1)/2$ nonresidues.

When we look at the residues of primes above, we see that some numbers appear more than once. For instance, 4 is residue of 5, 7, 11 and 13; 2, by contrast, appears only once; 3 appears twice. An interesting line of investigation could be, which prime moduli have a certain residue and which have not. The following function is a nice tool for this investigation:

```
hasResidue :: Integer → Integer → Bool
hasResidue n q   | q < 0 ∧ abs q > n  = hasResidue n (q `rem` n)
                 | q < 0               = hasResidue n (n + q)
                 | q ≡ 0               = True
                 | otherwise           = check 0
  where check x  | x ≡ n               = False
                 | (x ↑ 2) `rem` n ≡ q = True
                 | otherwise           = check (x + 1)
```

There is something special about this function that should be explained. First thing to notice is that it does not operate on *Natural*, but on *Integer* and, indeed, the first two guards immediately take care of negative residues (q). In the first line, a negative q with an absolute value greater than n is reduced to the negative remainder; for a negative remainder already reduced to a remainder modulo n, the function is simply called again on $n + q$, that is n minus the absolute value of q. This is *negative congruence* that we already encountered, when we started to discuss arithmetic modulo a prime. It is a way to generalise the case of $n - a$, where we are not looking for a fixed number, but for a residue defined relative to n, *e.g.* $n - 1$, which would just be -1.

For $q = 0$, the function simply yields *True*, since 0 is residue of any number. For positive integers, *hasResidue* calls *check* 0. *check*, as can be seen in the third line, counts the $x$es up to n. When it reaches n, it yields *False* (first line). Should it encounter a case where x^2 equals q mod n, it terminates yielding *True*.

We can test the function asking for 4 in 9, 15 and 21 and will see that in all three cases, the result is *True*. Now we would like to extend this to learn if all numbers starting from 5 (where it appears for the first time) have the residue 4:

$$
\begin{aligned}
&\mathit{haveResidue} :: \mathit{Integer} \to [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{haveResidue}\ \_\ [\,] = [\,] \\
&\mathit{haveResidue}\ q\ (n : ns) \mid n\ `\mathit{hasResidue}`\ q = n : \mathit{haveResidue}\ q\ ns \\
&\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \quad = \quad \mathit{haveResidue}\ q\ ns
\end{aligned}
$$

This function searches for numbers in a given list (second argument) that have q as a residue. We could, for instance, call this function on all numbers from 5 onwards (restricting the result to 10): *take* 10 (*haveResidue* 4 [5..]). The result is indeed $[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$.

Do not let yourself be confused by the fact that 4 is a quadratic residue of all numbers greater than 4. It is just trivial; in fact, all perfect squares are residues of numbers greater than these squares. The same is true for 9, which is residue of $10, 11, 12, \ldots$; 16 is residue of $17, 18, 19, \ldots$ and in general any number of the form x^2 is residue of any number $n > x^2$. This is just the definition of quadratic residue: $x^2 \equiv q \pmod{n}$, for the special case where $q = x^2$, which is trivially true, whenever $n > x^2$.

To continue the investigation into residues of primes, we specialise *haveResidue* to odd primes:

$$
\begin{aligned}
&\mathit{primesWithResidue} :: \mathit{Integer} \to [\mathit{Integer}] \\
&\mathit{primesWithResidue} = (`\mathit{haveResidue}`(\mathit{drop}\ 1\ \mathit{intAllprimes})) \\
&\quad \textbf{where}\ \mathit{intAllprimes} = \mathit{map}\ \mathit{fromIntegral}\ \mathit{allprimes}
\end{aligned}
$$

Since we have already seen that all numbers from 5 on have 4 as a residue, *primesWithResidue* 4, will just give us the primes starting from 5. A more interesting investigation is in fact -1, *i.e.* all primes p that have $p - 1$ as a residue:

primesWithResidue $(-1) = [5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97\ldots]$.

Is there something special about this list of primes? Not, perhaps, on the first sight. However, try this: *map* (`*rem*`4) (*primesWithResidue* (-1)) and you will see an endless list: $1, 1, 1, 1, 1, 1, \ldots$. In other words, all these primes are $\equiv 1 \pmod{4}$. If this is true, the following function should create exactly the same list of primes:

$$
\begin{aligned}
&\mathit{minus1Residues} :: [\mathit{Integer}] \\
&\mathit{minus1Residues} = \mathit{go}\ (\mathit{drop}\ 1\ \mathit{intAllprimes}) \\
&\quad \textbf{where}\ \mathit{go}\ [\,] \quad = [\,] \\
&\qquad\quad \mathit{go}\ (p : ps) \mid p\ `\mathit{rem}`\ 4 \equiv 1 = p : \mathit{go}\ ps \\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} \quad = \quad \mathit{go}\ ps \\
&\qquad\quad \mathit{intAllprimes} = \mathit{map}\ \mathit{fromIntegral}\ \mathit{allprimes}
\end{aligned}
$$

And, indeed, it does:

minus1Residues $= [5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97\ldots]$.

This fact is quite important. It is known as the *first supplement* to the *law of quadratic reciprocity*. We will start the proof of the first supplement with an apparently unrelated

theorem, namely *Wilson's Theorem*, which states that if and only if n is prime, then it holds that

$$(n-1)! \equiv -1 \pmod{n}. \tag{1.41}$$

To check this quickly with some small primes:

$$\begin{aligned} 2! = 2 &\equiv -1 \pmod{3} \\ 4! = 24 &\equiv -1 \pmod{5} \\ 6! = 720 &\equiv -1 \pmod{7} \\ 10! = 3628800 &\equiv -1 \pmod{11} \end{aligned}$$

and some small composites:

$$\begin{aligned} 3! = 6 &\equiv 2 \pmod{4} \\ 5! = 120 &\equiv 0 \pmod{6} \\ 7! = 5040 &\equiv 0 \pmod{8} \end{aligned}$$

The proof is rather simple. We first prove that Wilson's theorem holds for all primes, then we prove that it does not hold for composites.

To prove that it holds for primes, remember from modular arithmetic that, with a prime modulus n, every number $a \in 1 \ldots n-1$ has an inverse $a' \in 1 \ldots n-1$, such that $a \times a' = 1 \mod n$. For instance, the numbers $1 \ldots 6$ and their inverses modulo 7 are:

$$(1,1), (2,4), (3,5), (6,6)$$

For all pairs of numbers (a, a') where $a \neq a'$, the product of the pair is just 1. The example above seems to suggest that for all numbers, but 1 and $n-1$, $a \neq a'$. If this is true, then the factorial of $n-1$ modulo n, where n is prime, would translate into $1 \times 1 \times \cdots \times n-1$, which obviously is $\equiv (n-1) \pmod{n}$. But does it actually hold for all primes? Let us look: if a is its own inverse, *i.e.* $a = a'$, we would have $a^2 \equiv 1 \pmod{n}$. In other words: a^2 would leave a remainder of 1 divided by n. $a^2 - 1$, hence, should leave no remainder, thus: $a^2 - 1 \equiv 0 \pmod{n}$. We can factor $a^2 - 1$ into $(a+1)(a-1)$. The factors help us to find numbers that substituted for a would make the whole expression 0. One possibility is obviously 1: $(1+1) \times (1-1) = 2 \times 0 = 0$. Another possibility, however, is $n-1$:

$$(n-1+1)(n-1-1) =$$

$$n(n-2) = n^2 - 2n$$

and, since $n^2 - 2n$ contains only multiples of n:

$$n^2 - 2n \equiv 0 \pmod{n}.$$

This does not hold for any other number from the range $2 \dots n-1$, since there will be always a remainder that does not reduce to a multiple of n, for instance: $(n-2+1)(n-2-1) = (n-1)(n-3) = n^2 - 4n + 3$, which is congruent to 3 (mod n); $(2+1)(2-1) = 3 \times 1 = 3$, which, again, is congruent to 3 (mod n) and in general $(n-a+1)(n-a-1) = n^2 - 3an + a^2 - 1$, which, modulo n, is $a^2 - 1$. If $a \neq 1$, this is not congruent 0 (mod n). It therefore holds for all primes n that $(n-1)! \equiv -1 \pmod{n}$. □

Now the second part of the proof: That Wilson's theorem is never true when n is composite. We prove by contradiction and assume that there is a composite n, such that $(n-1)! \equiv -1 \pmod{n}$. That n is composite means that there is a prime number p that divides n. This p is one of the prime factors of n, *i.e.*: $n = mp$, for some integer m. This also means that p is smaller than n and in the range $2 \dots n-1$. Therefore, p must also divide $(n-1)!$, because it appears as one of the factors in $1 \times 2 \times \dots \times n-1$. In other words: $(n-1)! \equiv 0 \pmod{p}$.

But we also have $n = mp$. From modular arithmetic we know that if $a \equiv b \pmod{n}$, then also: $a \equiv b \pmod{mn}$. So, from $(n-1)! \equiv 0 \pmod{p}$, it follows that also $(n-1)! \equiv 0 \pmod{mp}$ and, since $n = mp$, $(n-1)! \equiv 0 \pmod{n}$. This contradicts our assumption that there is a composite n, such that $(n-1) \equiv -1 \pmod{n}$. □

Let us come back to the first supplement, which claims that -1 is a residue of an odd prime if and only if that prime is congruent 1 modulo 4. Any number not divided by 4 is either 1, 2 or 3 modulo 4. Since we are dealing only with odd primes, we can ignore the case 2, because no odd number will ever leave the remainder 2 divided by 4. That does only happen with even numbers not divided by 4, like 6, 10, 14, *etc.* We, hence, distinguish two cases: $p \equiv 1$ and $p \equiv 3$ both modulo 4. We first prove that an odd prime $p \equiv 1 \pmod{4}$ has residue -1 and then that an odd prime $p \equiv 3 \pmod{4}$ does not have residue -1.

We start with the observation that $p \equiv 1 \pmod{4}$ implies $(p-1) \equiv 0 \pmod{4}$, *i.e.* that $p-1$ is divisible by 4. This, in its turn, implies that we can group the remainders $1 \dots p-1$ into two sets with the same even number of elements. The remainders of the prime 5, for example, are $1, 2, 3, 4$ and we can group them into $\{\{1,2\},\{3,4\}\}$. For 13, these groups would be $\{\{1,2,3,4,5,6\},\{7,8,9,10,11,12\}\}$.

Now we rewrite the second group in terms of negative congruences:

$$\{\{1,2,3,4,5,6\},\{-6,-5,-4,-3,-2,-1\}\}$$

and then organise the groups as pairs of equal absolute values:

$$\{(1,-1),(2,-2),(3,-3),(4,-4),(5,-5),(6,-6)\}.$$

To compute the factorial of $p-1$, we could first multiply the members of each pair: $\{-1,-4,-9,-16,-25,-36\}$. It is essential to realise that the number of negative signs is even because the number of elements of each group is even. They, hence, cancel out on multiplication. This would be the same as squaring the members of the first group (1 to 6) before multiplying them: $\{1,4,9,16,25,36\}$. We can do this the other way round as well: first, we multiply the two halfs out, creating the factorial for each group, and then multiply the two equal results, which is the same as squaring one of the results. In other words, if $p \equiv 1 \pmod 4$, then

$$(p-1)! = \left(\frac{p-1}{2}\right)!^2. \tag{1.42}$$

The right-hand side of this equation is a formal description of what we did above. We split the numbers $1 \ldots p-1$ into halves: $1 \ldots \frac{p-1}{2}$ and $-\frac{p-1}{2} \cdots -1$, computed the factorial of each half and then multiplied the results, which of course are equal and multiplying them is thus equivalent to squaring.

For a prime p with the residue -1, there must be one number a, such that $a^2 \equiv -1 \pmod p$. The equation above shows that $(\frac{p-1}{2})!^2$ is actually $(p-1)!$, which, according to Wilson's theorem, is $-1 \pmod p$. $\frac{p-1}{2}!$, which squared is $(p-1)!$ and, according to Wilsons' theorem, -1. $\frac{p-1}{2}!$ is therefore such a number a. This, as shown above, is the case, if we can split the sequence of numbers $1 \ldots p-1$ into two halves with an even number of members each. This, however, is only possible for an odd prime p, if $p-1 \equiv 0 \pmod 4$, which implies that $p \equiv 1 \pmod 4$. □

We will now show that primes of the form $p \equiv 3 \pmod 4$ do not have the residue -1 to complete the proof. Let us assume there is a number a, such that $a^2 \equiv -1 \pmod p$ and $p \equiv 3 \pmod 4$.

We start with the equation

$$a^2 \equiv -1 \pmod p \tag{1.43}$$

and raise both sides to the power of $\frac{p-1}{2}$:

$$a^{2\frac{p-1}{2}} \equiv -1^{\frac{p-1}{2}} \pmod p. \tag{1.44}$$

This can be simplified to

$$a^{p-1} \equiv -1^{\frac{p-1}{2}} \pmod{p}. \tag{1.45}$$

Note that $p-1$ is even (since p is an odd prime). But, since it is not divisisble by 4, since otherwise $p \equiv 1 \pmod 4$, $(p-1)/2$ must be odd. An example is $p = 7$, for which $(p-1)/2 = 3$. -1 raised to an odd power, however, is -1 and therefore we have:

$$a^{p-1} \equiv -1 \pmod{p}. \tag{1.46}$$

But that cannot be true, because Fermat's little theorem states that

$$a^{p-1} \equiv 1 \pmod{p} \tag{1.47}$$

There is only one p for which both equations are true at the same time, namely 2: $a^1 \equiv 1 \pmod 2$. This is actually true for any odd a.

But we are looking at odd primes and 2 is not an odd prime. Therefore, one of the equations must be wrong. Since we know for sure that Fermat's theorem is true, the wrong one must be 1.46. Therefore, -1 cannot be a residue of primes of the form $p \equiv 3 \pmod 4$ □.

There is also a *second supplement* to the law of reciprocity, which happens to deal with 2 as residue. When we look at these numbers, using *primesWithResidue* 2, we get: 7, 17, 23, 31, 41, 47, 71, 73, 79, 89, 97, 103,... What do these primes have in common? They are all one off numbers divisible by 8:

$$\begin{aligned} 7 &\equiv -1 \pmod 8 \\ 17 &\equiv 1 \pmod 8 \\ 23 &\equiv -1 \pmod 8 \\ 31 &\equiv -1 \pmod 8 \\ 41 &\equiv 1 \pmod 8 \\ 47 &\equiv -1 \pmod 8 \\ &\dots \end{aligned}$$

The second supplement indeed states that ± 2 is residue of an odd prime p if and only if $p \equiv \pm 1 \pmod 8$.

We will not prove this theorem here. Instead, we will look at a general criterion to decide quickly whether a number is residue of an odd prime, namely *Euler's Criterion*, which states that, if p is an odd prime, then:

$$a^{\frac{p-1}{2}} \equiv 1 \pmod{p} \text{ iff } a \text{ is a residue of } p \tag{1.48}$$

$$a^{\frac{p-1}{2}} \equiv -1 \pmod{p} \text{ iff } a \text{ is a nonresidue of } p \tag{1.49}$$

We can translate this criterion into Haskell as:

```
euCriterion :: Integer → Integer → Bool
euCriterion a p | a < 0 ∧ abs a > p = euCriterion (a `rem` p) p
                | a < 0             = euCriterion (p + a) p
                | otherwise         = let n = (p − 1) `div` 2
                                      in (a ↑ n) `rem` p ≡ 1
```

As we did before, we first handle the cases of negative congruence: a negative number is reduced to a negative number with an absolute value in the range of $1 \ldots p-1$ and then p is added. A positive number is just raised to the power of $(p-1)/2$. If the remainder of this number is 1, this number is indeed a residue of p.

You see that in Euler's Criterion there appears a formula that we already know from the first supplement and, indeed, the proof of the Criterion with the background of the first supplement is quite simple.

We start by considering the case where a is a nonresidue: $a^{\frac{p-1}{2}} \equiv -1 \pmod{p}$. For equations of the form $bx \equiv a \pmod{p}$, as we have discussed, when we introduced arithmetic modulo a prime number, there is a unique solution b' such that $b \times b' \equiv a \pmod{p}$. Notice that $b \neq b'$, since, otherwise we would have the case $b^2 \equiv a \pmod{p}$ and, in consequence, a would be a residue of p, but we are discussing the case that a is a nonresidue. We now apply the same technique as above: we build pairs of bs and b's to simplify the computation of the factorial. For each pair (b, b'), where $b \neq b'$, we have $b \times b' \equiv a \pmod{p}$. Since there is exactly one b' for any b in $1 \ldots p-1$, there are $\frac{p-1}{2}$ pairs of (b, b'). For instance:

$1 \times 6 \equiv 6 \pmod{7}$
$2 \times 3 \equiv 6 \pmod{7}$
$4 \times 5 \equiv 6 \pmod{7}$.

When we compute the factorial, we multiply these pairs out, each of which gives a. So, the factorial is $a \times a \times \cdots \times a$ and, since there are $(p-1)/2$ such pairs, $a^{\frac{p-1}{2}}$:

$$(p-1)! \equiv a^{\frac{p-a}{2}} \pmod{p}. \tag{1.50}$$

From Wilson's theorem, we know that $(p-1)! \equiv -1 \pmod{p}$. We, therefore, conclude that $a^{\frac{p-a}{2}} \equiv -1 \pmod{p}$. □

Now we consider the other case, *i.e.* that a is a quadratic residue. In this case, we actually have a solution for $x^2 \equiv a \pmod{p}$.

Consider Fermat's little theorem again:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{1.51}$$

We subtract 1 from both sides:

$$a^{p-1} - 1 \equiv 0 \pmod{p} \tag{1.52}$$

and force in our magic formula $\frac{p-1}{2}$ by factoring the left-hand side. What we are doing step-by-step is

$$a^{\frac{p-1}{2}} \times a^{\frac{p-1}{2}} = a^{\frac{p-1}{2}+\frac{p-1}{2}} =$$
$$a^{\frac{(p-1)+(p-1)}{2}} = a^{\frac{2p-2}{2}} = a^{p-1}.$$

We can reformulate equation 1.52 accordingly:

$$(a^{\frac{p-1}{2}} - 1)(a^{\frac{p-1}{2}} + 1) \equiv 0 \pmod{p}. \tag{1.53}$$

Since, to make a product 0, one of the factors must equal 0, $a^{\frac{p-1}{2}}$ must take either the value 1 or -1.

If a is a quadratic residue, then we have some integer x, such that $a \equiv x^2$. We, hence, could write:

$$(x^{2^{\frac{p-1}{2}}} - 1)(x^{2^{\frac{p-1}{2}}} + 1) \equiv 0 \pmod{p}. \tag{1.54}$$

That would mean that, to make the first factor 0, x^2 must be 1 modulo p and, to make the second factor 0, x^2 must be -1 modulo p. We, hence, want either:

$$x^{2^{\frac{p-1}{2}}} \equiv 1 \pmod{p} \tag{1.55}$$

or

$$x^{2^{\frac{p-1}{2}}} \equiv -1 \pmod{p} \tag{1.56}$$

Since x^{a^b} is just x^{ab}, we can simplify to:

$$x^{p-1} \equiv 1 \pmod{p}, \tag{1.57}$$

for equation 1.55 and

$$x^{p-1} \equiv -1 \pmod{p}, \tag{1.58}$$

for equation 1.56. The second result contradicts Fermat as already seen above and, thus, cannot be the case. One may be tempted to say immediately that the second factor cannot be 0, since then we would have $x^2 = -1$, which cannot be true for any integer x. With modular arithmetic, however, this is not true. A counterexample is $2^2 = 4$, which is -1 modulo 5. We therefore need the reference to Fermat's little theorem to actually show that equation 1.56 leads to a contradiction with a proven theorem.

The simplification of equation 1.55, however, just yields the little theorem. We, thus, can derive Euler's criterion directly from Fermat and that proves that, if a is a quadratic residue of p, we always have $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. □

The French mathematician Adrien-Marie Legendre (1752 – 1833) defined a function using Euler's Criterion and a nice notation to express this function known as the *Legendre Symbol*. It can be defined as:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}. \tag{1.59}$$

For $a \in \mathbb{Z}$ and p an odd prime, $\left(\frac{a}{p}\right) \in \{1, -1, 0\}$. More specifically, if a is a residue of p, then $\left(\frac{a}{p}\right)$ is 1, if it is a nonresidue, it is -1, and if $a \equiv 0 \pmod{p}$, it is 0.

The Legrende Symbol has some interesting properties. We will highlight only two of them here. The first is that if $a \equiv b \pmod{p}$, then (trivially)

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right). \tag{1.60}$$

More interesting is multiplicativity:

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right). \tag{1.61}$$

For example, $\left(\frac{3}{11}\right) = 1$ and $\left(\frac{5}{11}\right) = 1$. So $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = 1$ and $\left(\frac{3\times5=15}{11}\right)$ is 1 as well. $\left(\frac{6}{11}\right) = -1$ and $\left(\frac{3\times6=18}{11}\right)$ is -1. $\left(\frac{2}{11}\right) = -1$ and $\left(\frac{2\times6=12}{11}\right) = 1$. A final example with 0: $\left(\frac{22}{11}\right) = 0$ and, with any other number, *e.g.*: $\left(\frac{2\times22=44}{11}\right) = 0$.

We can implement the Legendre Symbol easily in Haskell as:

$$
\begin{array}{l}
\mathit{legendre} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{legendre}\ a\ p = \textbf{let}\ n = (p-1)\ `\mathit{div}`\ 2 \\
\qquad\qquad\qquad\qquad \textbf{in case}\ (a \uparrow n)\ `\mathit{rem}`\ p\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad 0 \to 0 \\
\qquad\qquad\qquad\qquad\qquad 1 \to 1 \\
\qquad\qquad\qquad\qquad\qquad x \to x - p
\end{array}
$$

For some time now, we are beating around the *law of quadratic reciprocity* and it appears to be high time to finally explain what this law is all about. The law is about two primes, p and q, and claims that if the product

$$\frac{p-1}{2} \times \frac{q-1}{2} = \frac{(p-1)(q-1)}{4}$$

is even, then, if p is a residue of q, q is also a residue of p. Otherwise, if the above product is odd, then, if p is a residue of q, q is nonresidue of p.

This can be formulated much more clearly using the Legendre symbol:

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}} \tag{1.62}$$

If $\frac{(p-1)(q-1)}{4}$ is even, then the right-hand side of the equation becomes 1, otherwise it is -1. To become 1, the Legendre Symbols on the left-hand side of the equation must be either both negative or both positive. They are both positive, namely 1, if p is residue of q and q is residue of p. They are both negative, namely -1, if neither p is residue of q nor q of p.

For the right-hand side to become -1, one of the Legendre Symbols must be negative and the other positive. This is the case if either p is residue of q, but q is not residue of p or if q is a residue of p, but p is not a residue of q.

For example, look at the primes 7 and 11. The residues of 7 are $\{0, 1, 2, 4\}$ and, since $11 \equiv 4 \pmod 7$, 11 is a residue of 7. The residues of 11 are $\{0, 1, 3, 4, 5, 9\}$. 7, hence, is not a residue of 11. Now look at the fraction

$$\frac{(7-1)(11-1)}{4} = \frac{60}{4} = 15,$$

which is odd. Therefore 7 can only be a residue of 11, if 11 is a nonresidue of 7 and vice versa. 11 is a residue of 7, therefore 7 is not a residue of 11.

What about 7 and 29? Look at the magic fraction:

$$\frac{(7-1)(29-1)}{4} = \frac{168}{4} = 42.$$

42 is even, therefore 7 can only be residue of 29 if 29 is a residue of 7. The residues of 29 are: $\{0, 1, 4, 5, 6, 7, 9, 13, 16, 20, 22, 23, 24, 25, 28\}$. Since 7 is included (and 7 hence is a residue of 29), 29 must also be a residue of 7. Since $29 \equiv 1 \pmod 7$ and 1 is indeed residue of 7, 29 is a residue of 7.

The residues of 5 are $\{0, 1, 4\}$. For 5 and 7, the magic formula is even:

$$\frac{(5-1)(7-1)}{4} = \frac{24}{4} = 6.$$

So, since 5 is a nonresidue of 7, 7 must also be a nonresidue of 5.

The law had already been conjectured by Euler and Legendre, when Gauss finally proved it in the *Disquisitiones*. Gauss called the theorem the *Golden Rule* and, interestingly, the *fundamental theorem of arithmetic* highlighting the value he attached to it. During his life he provided eight different proofs. Many more proofs have been devised since Gauss. According to the *Book*, there were 196 different proofs in the year 2000. We will not go through them here.

1.9 Generators and Subgroups

Let us look at powers of numbers modulo a prime from another angle. In the previous section, we looked at remainders that are squares. Now we look at what happens to remainders, when we raise them to exponents:

$$x^1, x^2, x^3, \ldots, x^{p-1} \mod p.$$

What do we expect to happen? We first can predict that, for any number x, there is an exponent k, such that $x^k = 1$. In other words, the set that we create in this way always contains the identity. One case is x^{p-1} for which we know from Fermat's little theorem that it is congruent to 1 for any number x. For instance $3^6 = 1 \pmod 7$. We also know that, if x is a residue, then $x^{\frac{p-1}{2}} = 1$. There, hence, are numbers that result in a smaller set of numbers, since, once $x^k = 1$, the sequence will just repeat with $x^{k+1} = x$, $x^{k+2} = x^2$ and so on.

If x is a nonresidue, we know that $x^{\frac{p-1}{2}} = -1$. Then, $x^{\frac{p-1}{2}} x^{\frac{p-1}{2}} = x^{p-1} = 1$. Since, in the group of any odd prime there are residues and nonresidues, we know for sure that some numbers create the whole group and others do not.

For 7, the powers of 3, for instance, yield the whole group:

$$
\begin{aligned}
3^1 &\equiv 3 \pmod 7 \\
3^2 &\equiv 2 \pmod 7 \\
3^3 &\equiv 6 \pmod 7 \\
3^4 &\equiv 4 \pmod 7 \\
3^5 &\equiv 5 \pmod 7 \\
3^6 &\equiv 1 \pmod 7
\end{aligned}
$$

The powers of 4, which is a residue of 7, do not:

$$
\begin{aligned}
4^1 &\equiv 4 \pmod 7 \\
4^2 &\equiv 2 \pmod 7 \\
4^3 &\equiv 1 \pmod 7.
\end{aligned}
$$

Second, we observe that we create a set of numbers with certain relations among them:

$$
\begin{aligned}
a &= x^1 \\
b &= x^2 \\
c &= x^3 \\
&\dots \\
1 &= x^{p-1}
\end{aligned}
$$

Any multiplication of two numbers in the set results in another number in the set. Therefore, any power of a number in the set will result in another number in the set. Since $b = aa$ and $c = aaa$, it also holds that $c = ab$. We can go on this way by observing that every number n_i in the set is the result of multiplying the first number in the set a, which is just $x^1 = x$, with its predecessor n_{i-1} or the second number $x^2 = aa$ with n_{i-2} and so on. The set, hence, is closed under multiplication. Furthermore, at some step, n_i becomes 1 and, from any number in the set, we can get to 1 by multiplying another number in the set. This is trivially true for $1 \times x^k = 1$, if k is the number such that $x^k \equiv 1 \pmod n$; it is also true for $x^1 x^{k-1} = 1$ and it is in general true for any number $x^i x^{k-i}, 0 \leq i \leq k$. When we have, for instance $k = 3$, then $1 \times aaa = aaa = 1$. $a \times aa = aaa = 1$ and $aaa \times aaa = 1 \times 1 = 1$. In other words, for every element a in the set, there is also its inverse a' in the set, such that $aa' = 1$. That means that the resulting set is again a multiplicative group.

We call a number that generates a group G, a *generator* of G. It is often also called a *primitive element* of G. If the group H generated by a number g modulo p is not the

whole group G of p, *i.e.* the numbers $\{1, \ldots, p-1\}$, then we call H a *proper subgroup* of G. A subgroup H of a group G is a group that contains only numbers that are also in G. G, hence, is a subgroup of G itself. A proper subgroup, H, of a group G is a subgroup, where not all members of G are also in H. A proper subgroup G is therefore smaller than G. The group generated by the generator 3 modulo 7, for instance, is a subgroup of G (it is in fact identical to G). The group generated by the generator 4 modulo 7, too, is a subgroup of G, but it is a proper subgroup, since all elements in this group are also in G, but not all elements in G are in this subgroup. This is the same concept as the subset in set theory.

We can devise a simple function to generate a group, given p, the prime, and g, the generator:

$$\begin{aligned}&\mathit{generate} :: \mathit{Natural} \to \mathit{Natural} \to [\mathit{Natural}]\\&\mathit{generate}\ p\ g = \mathit{sort}\ (\mathit{nub}\ (\mathit{map}\ (\lambda a \to (g \uparrow a)\ `\mathit{rem}`\ p)\ [1\,..\,p-1]))\end{aligned}$$

Note that we *nub* the result to restrict the resulting set to the group itself. For the case where g generates a proper subgroup of the entire group, we otherwise would get repetitions. We also *sort* the groups to get a canonical order, *i.e.* $[1, 2, 4]$ instead of $[4, 2, 1]$.

Let us look at the groups generated by all the numbers $\{1 \ldots 6\}$ modulo 7:

$\mathit{generate}\ 7\ 1 = [1]$
$\mathit{generate}\ 7\ 2 = [1, 2, 4]$
$\mathit{generate}\ 7\ 3 = [1, 2, 3, 4, 5, 6]$
$\mathit{generate}\ 7\ 4 = [1, 2, 4]$
$\mathit{generate}\ 7\ 5 = [1, 2, 3, 4, 5, 6]$
$\mathit{generate}\ 7\ 6 = [1, 6]$.

We see 4 different groups. Two of these groups are quite trivial: $g = 1$ generates a group with just 1 element, since $1 \times 1 = 1$; $g = 6$ generates a group with two elements, since $1 \times 6 = 6$ and $6 \times 6 = 1$. These two trivial groups exist for any prime greater 2, since $1 \times 1 = 1$, trivially, holds for any modulus and $(p-1)(p-1) = 1$ holds for any prime modulus. 2 is an exception, because, with 2, we have $1 = p - 1$ and, therefore, 2 has only one trivial group.

The other subgroups modulo 7 are: $\{1, 2, 4\}$ generated by 2 and 4 and the complete group $\{1 \ldots p-1\}$ generated by 3 and 5. The size of these groups are 1 and 2 (for the trivial groups) and 3 and 6 for the non-trivial ones. We call the size of a group its *order* and write $|G|$ for the order of group G. The order of the complete prime group is, as we know, $p - 1$. What about the order of the other groups? Is there a pattern too?

To further investigate, we define a function that shows all subgroups of a prime:

$$\begin{aligned}&\mathit{allGroups} :: \mathit{Natural} \to [[\mathit{Natural}]]\\&\mathit{allGroups}\ p = \mathit{map}\ (\mathit{generate}\ p)\ [2\,..\,p-2]\end{aligned}$$

Note that we leave out the trivial groups 1 and $\{1, p-1\}$; we know that they exist for any p, so there is not much information added by showing them.

These are the results for *allGroups* 13 (with duplicates already removed):

$[1,2,3,4,5,6,7,8,9,10,11,12]$
$[1,3,9]$
$[1,3,4,9,10,12]$
$[1,5,8,12]$

We see again 4 groups. They have the orders 3, 4, 6 and 12. There are also the two trivial groups with order 1 and 2 of course. A striking peculiarity is that for both 7 and 13, all subgroups have orders that divide the order of the main group. For 7, the order of the main group is 6 and the proper subgroups have the orders 1, 2 and 3. For 13, the order of the main group is 12 and the proper subgroups have the orders 1, 2, 3, 4 and 6. We will use the following function to investigate this further:

$$orders :: Natural \to [Int]$$
$$orders = nub \circ map\ length \circ allGroups$$

The result for *orders* 13 is $[12,3,6,4]$.

We now map this function on the primes; we drop the first two primes, 2 and 3, because they have only the trivial subgroups and start by looking at the first 9 primes starting with 5 calling *map orders* (*take* 9 (*drop* 2 *allprimes*)):

5	7	11	13	17	19	23	29	31
4	3,6	5,10	3,4,6,12	4,8,16	3,6,9,18	11,22	4,7,14,28	3,5,6,10,15,30

The suspicion is confirmed: The orders of the subgroups always divide the order of the prime group. Indeed, this fact is known as *Lagrange's theorem*. It was proven by Joseph Louis Lagrange, 1736 – 1813, an Italian mathematician who lived and taught in Turin, Berlin and Paris. Lagrange proved another important theorem that we already met: Wilson's theorem.

Lagrange's theorem is a kind of crossroads between different branches of mathematics including group theory, number theory, set theory and algebra. It is as such a quite deep theorem and to appreciate its full meaning, we need much more mathematical machinery than we have available right now. We will come back to Lagrange's theorem and provide a complete proof. Here, we will provide a quite simple proof sufficient for the current context concerning finite multiplicative groups. But even this proof provides surprisingly deep insight into the structure of groups.

Lagrange's theorem states that for any group G and any of its subgroups H: $|H|$ divides $|G|$. We start the proof by considering an arbitrary group modulo a prime. Such a group is generated by a sequence of powers of a: $a^1, a^2, \ldots, a^k$, where $a^k = 1$. For sake of explicitness, let us consider a concrete example, say, the group modulo 7, which has order 6. Let the sequence of powers of a, any primitive element of that group, be the

sequence:

$$a, b, p-1, b', a', 1.$$

In this group, the placement of b and its inverse b' is arbitrary. The placement of $a = a^1$, $p-1$, a' and 1, however, is on purpose and respects the order in which these numbers necessarily appear, when the numbers reflect the numbers generated by a^1, a^2 and so on. The first number, a^1, trivially is a. The last number a^k is 1. The last but one number is that number in the group that multiplied by a results in 1, *i.e.* the inverse of a. Since a^6, in this example, is 1, a^3 must be its own inverse, *i.e.* a number that multiplied by itself, $a^{3+3} = a^6$, is 1. We know that $p-1$ is the only number, besides 1 itself of course, that is its own inverse.

Since the sequence terminates with $a^6 = 1$, it would repeat with $a^7 = a$, when we continue. We show this in the following table up to exponent $k = 12$:

1	2	3	4	5	6	7	8	9	10	11	12
a	b	p-1	b'	a'	1	a	b	p-1	b'	a'	1

Based on this information, try to imagine the group generated by b. b is aa, so we get (with the headline indicating the exponents of a, not of b!):

1	2	3	4	5	6	7	8	9	10	11	12
	b		b'		1		b		b'		1

Indeed, we know that $b = a^2$ and that $b' = a^4$. Consequently, $bb = aaaa = a^4 = b'$ and, even further, $bbb = a^6 = 1$. We therefore see a group with three members.

The inverse of b would generate another group with three members, but since the exponent of b', which is 4, does not divide the exponent of 1, 6, we need more than one cycle to terminate the group:

1	2	3	4	5	6	7	8	9	10	11	12
			b'				b				1

Continuing this scheme, we can easily imagine the group generated by $p-1$:

1	2	3	4	5	6	7	8	9	10	11	12
		p-1			1			p-1			1

and that generated by 1:

1	2	3	4	5	6	7	8	9	10	11	12
					1						1

In other words, the periodicity of element 1, which appears every 6 as, and the relation among numbers, that is $b = aa$ and $p - 1 = aaa$ in this example, determine all possible group orders. If $a^i = p - 1$, then $a^{2i} = 1$ determines the order of the group generated by a. If b is member of that group and $b = a^j$, then j must divide either $2i$ or a multiple of that number – a', which is a^5, for instance, would only return to 1 at index 30. Since only every j^{th} element is in the group of b, the group has $\frac{lcm(2i,j)}{j}$ members. For b, we have $j = 2$ and $\frac{lcm(6,2)}{2} = 3$ group members. For b', we have $j = 4$ and $\frac{lcm(6,4)}{4} = 3$ group members. For $p - 1$, we have $j = 3$ and $\frac{lcm(6,3)}{3} = 2$ group members. For 1, we have $j = 6$ and $\frac{lcm(6,6)}{6} = 1$ group member. For a' we have $j = 5$ and $\frac{lcm(6,5)}{5} = 6$ group members. For a, finally, we have $j = 1$ and $\frac{lcm(6,1)}{1} = 6$ group members. The number $\frac{lcm(2i,j)}{j}$, for sure, divides $2i$, since $lcm(2i, j)$ is a multiple of both, $2i$ and j. That is all what we wanted to prove. □

For our case at hand, this proof (even though a bit sloppy using an example) is sufficient. The theorem, however, is not limited to remainders of primes. We will see other examples soon and, indeed, we already saw an example with very similar effects. In the previous section, we discussed composition of permutations and there we saw that a permutation with orbits of different size n and m, need $lcm(n, m)$ applications to come back to the original sequence.

But let us come back to problems of primes. When we look at the table of subgroup orders above, we see some primes, such as 11 and 23, with strikingly fewer subgroups than the primes in their surrounding. 11 and 23, both, have two subgroups, while 13, 19 and 29 all have four subgroups. Is this a pattern or is it just one of the curiosities that arise with small numbers? Here is a list of six more primes generated by *orders* (*take* 6 (*drop* 11 *allprimes*)):

37	41	43	47	53	59
3,4,6,9,12,18,36	4,5,8,10,20,40	3,6,7,14,21,42	23,46	4,13,26,52	29,58

Most primes we see in this sequence have 6 or 7 subgroups and one, 53, has 4. The outliers are 47 and 59 with 2 subgroups each. So, what is special about the primes 11, 23, 47 and 59?

Let us examine their subgroup orders:

11	23	47	59
5,10	11,22	23,46	29,58

All of these primes have the subgroup with order $p - 1$ and a subgroup with order $\frac{p-1}{2}$, *i.e.* the half of the order of the main group. In all cases, the half of the order of the main group is again a prime number: 5, 11, 23 and 29. When the order of the main group has only two prime factors, namely 2 and q, where q is again prime, then,

since the order of any subgroup must divide the order of the main group, there cannot be another subgroup besides the trivial ones with order 1 and 2. This fact has huge importance for cryptography, especially the Diffie-Hellman key exchange protocol and the Schnorr signature. These cryptosystems make use of primes of the form $2p+1$ where p is also prime to guarantee that any element chosen but 1 and $p-1$ is member of a huge subgroup. We will discuss this later in this chapter.

Primes of the form $2p+1$ (with p prime) are called *safe primes*. The other prime, the p in the safe prime formula, is called *Sophie Germain* prime after the great French mathematician Sophie Germain (1776 – 1831). Germain began to study math with about 13 years of age. Later, when the École Polytechnqiue opened in Paris during the French Revolution, she started to send essays to the teachers there and one of them, again Lagrange, recognised her talent.

Since it was not allowed for women to study at the Polytechnique at that time, Germain used the name of a former student, Antoine-August Leblanc, when she presented her papers. Lagrange, however, was quite excited about the quality of these essays and was eager to get to know this talented student. So, Germain was forced to reveal her identity, when Lagrange invited the person he believed to be Leblanc. Fortunately, Lagrange continued to support Germain and she was able to present important results in mathematics including number theory and won prestigious prizes from the Paris Academy of Sciences.

Germain would also correspond with Gauss, again under the name Leblanc; when the Napoleon army occupied Braunschweig, where Gauss lived at that time, she asked a friend of the family who was actually a general of the French army to see after Gauss' safety during the occupation. On this occasion, Gauss learnt who his French correspondent was and wrote later:

> How can I describe my astonishment and admiration on seeing my esteemed correspondent M leBlanc metamorphosed into this celebrated person... when a woman, because of her sex, our customs and prejudices, encounters infinitely more obstacles than men in familiarising herself with knotty problems, yet overcomes these fetters and penetrates that which is most hidden, she doubtless has the most noble courage, extraordinary talent, and superior genius.

Let us devise a fuction in the honour of Sophie Germain to list the primes that bear her name:

$$sophieprimes :: [Natural]$$
$$sophieprimes = filter\ (\lambda p \rightarrow prime\ (2 * p + 1))\ allprimes$$

The first 16 Sophie Germain primes, listed with *take* 16 *sophieprimes* are:

2, 3, 5, 11, 23, 29, 41, 53, 83, 89, 113, 131, 173, 179, 191, 233.

Let us confirm that the primes of the form $2p+1$ that correspond to these Sophie

Germain primes all have only two non-trivial groups. We use the function:

$$safeprimes :: [Natural]$$
$$safeprimes = map\ (\lambda q \to 2 * q + 1)\ sophieprimes$$

These are 16 safe primes greater than 59 listed with $take\ 16\ (dropWhile\ (\leqslant 59)\ safeprimes)$:

83, 107, 167, 179, 227, 263, 347, 359, 383, 467, 479, 503, 563, 587, 719, 839.

The subgroup orders of the first 9 of them are:

83	107	167	179	227	263	347	359	383
41,82	53,106	83,166	89,178	113,226	131,262	173,346	179,358	191,358

and we see that, indeed, all safe primes have exactly two non-trivial groups.

The concept can be extended to primes of the form $p = nq+1$ where n is even. This way we obtain a p with potentially much more subgroups than just the group of order $\frac{p-1}{2}$, *viz.* one subgroup per prime factor of n. The group we want to use, that of size $\frac{p-1}{n}$, is called the *Schnorr group* of p after the German mathematician and cryptographer Claus Schnorr, who studied the mathematical properties of this group extensively and invented a cryptographic authentication system based on it, the *Schnorr signature*. We will come back to that soon.

1.10 Primality Tests

Primality tests are algorithms that test whether a given number is prime. Until now, we have only one primality test. This test, basically, creates one prime after the other until it finds one that divides the number in question. If the first number found is that number itself, then this number is prime. The test finds an answer after having, in the worst case, examined $\sqrt{n}$ numbers. For small numbers, this is great. For large numbers, however, this may turn out to be very expensive. To be honest, in spite of all its ingenuity, this algorithm is quite shabby. It is just a brute force attack that tries to solve the problem by looking at all primes that stand between us and $\sqrt{n}$.

In the previous sections, we have learnt a lot of facts concerning prime numbers. Perhaps some of those facts may help us to distinguish between composites and primes. The first candidate is Fermat's little theorem, which states that, for any integer a and any prime p:

$$a^{p-1} \equiv 1 \pmod{p}. \tag{1.63}$$

This way, Fermat's theorem provides a criterion for a number being prime (or, more precisely, being not prime) that can be easily tested. Power is still a heavy operation with large numbers, but we need to apply the opertation only once and that is indeed

much cheaper than going through millions of primes to test just one number for primality. A test based on Fermat could be implemented like this:

$$
\begin{aligned}
&\mathit{fprime} :: \mathit{Natural} \to \mathit{Bool}\\
&\mathit{fprime}\ 0 = \mathit{False}\\
&\mathit{fprime}\ 1 = \mathit{False}\\
&\mathit{fprime}\ 2 = \mathit{True}\\
&\mathit{fprime}\ p = (2 \uparrow (p-1))\ \text{`}\mathit{rem}\text{`}\ p \equiv 1
\end{aligned}
$$

Since the condition should hold for any integer a, we just choose 2 and check if it, raised to $p-1$, leaves the remainder 1 divided by p. (We do not choose 1, of course, since 1 trivially leaves 1 with any number p.) We could test this new primality test with the old one comparing their results, like this: $[(n, \mathit{prime}\ n, \mathit{fprime}\ n) \mid n \leftarrow [1..]]$. The result look like this:

(1, *False*, *False*)
(2, *True*, *True*)
(3, *True*, *True*)
(4, *False*, *False*)
(5, *True*, *True*)
...

and without much surprise, we see that *fprime* produces the same results as *prime*. But be careful: Fermat's little theorem claims that all primes adhere to the rule, but it does not make any statement on composites. According to the theorem, composites may or may not leave a remainder of 1 with a^{p-1}. What the criterion establishes is therefore not that p is prime, but that, if the condition is not fulfilled, p is not prime. Is this relevant? Well, let us see what happens, when we continue the listing above:

...
(339, *False*, *False*)
(340, *False*, *False*)
(341, *False*, *True*)
...

The primality tests disagree on 341! Which one is right? First let us look at the Fermat test:

$$2^{340} \equiv 1 \pmod{341}.$$

According to this test, 341 appears to be prime. So it must not have any factors. However, *trialfact* 341:

$[11, 31]$

Apparently, 341 has two factors, 11 and 31, since $11 \times 31 = 341$. So, 341 is definitely

not a prime. In fact, there are composites that pass certain primality tests, so called *pseudoprimes*. We could have actually avoided falling into this trap by chosing a different a, for instance:

$$3^{340} \equiv 56 \pmod{341}.$$

However, there are still 98 of 338 numbers in the range $3 \ldots 340$ for which the Fermat test would have succeeded. We could repair the Fermat test by making it stronger: we could demand that all numbers $2 \ldots 340$ must pass the test, before we accept p being prime. This, however, would make the Fermat test quite expensive – and that it is inexpensive was the main reason we have chosen it in the first place. As an alternative, we could demand that there should be a certain amount of numbers for which the Fermat test should not fail. But even this would not help a lot, since there are numbers where indeed most $a \in \{2 \ldots p-1\}$ would pass the test, namely, the *Carmichael numbers*:

561, 1105, 1729, 2465, 2821, 6601, 8911, ...

A Carmichael number n is a composite number such that $a^{n-1} \equiv 1 \pmod{n}$ for every a coprime to n. When Robert Carmichael discovered the first of these numbers in 1910, the notion already existed, but under another name and with another definition. Already in 1899, the German mathematician Alwin Korselt defined numbers n, such that n is squarefree (no prime factor appears more than once in the prime factorisation of that number) and that, for every prime factor p, it holds that $(p-1)|(n-1)$. It turned out that both definitions are equivalent. 561, for example, has the factorisation $\{3, 11, 17\}$. Trivially, $3-1=2$ divides $561-1=560$; $11-1=10$ also divides 560 and, finally, $17-1=16$ divides 560, since $16 \times 35 = 560$.

Numbers coprime to 561 in the range $1 \ldots 560$ are all numbers not multiples of the prime factors 3, 11 or 16. 242 of the 560 numbers $1 \ldots 560$ are actually multiples of (at least) one of the prime factors. All other numbers are coprime to 561. In other words, more than half of the numbers will pass the Fermat test. Carmichael numbers are therefore hard to distinguish from primes by means of tests that avoid testing all remainders of n.

There are candidates for much stronger primality tests, however. Wilson's theorem, for instance, provides a criterion that holds for primes only, namely:

$$(p-1)! \equiv -1 \pmod{p}. \tag{1.64}$$

A primality test that would not fall for Carmichael numbers and other pseudoprimes could be based on Wilson's theorem:

$$
\begin{aligned}
&\mathit{wprime} :: \mathit{Natural} \to \mathit{Bool}\\
&\mathit{wprime}\ 0 = \mathit{False}\\
&\mathit{wprime}\ 1 = \mathit{False}\\
&\mathit{wprime}\ 2 = \mathit{True}\\
&\mathit{wprime}\ n = (\mathit{fac}\ (n-1))\ `\mathit{rem}`\ n \equiv (n-1)
\end{aligned}
$$

And, indeed, *wprime* gives *False* for 341 as it does for any of the Carmichael numbers. Unfortunately, factorial is a very expensive operation rendering *wprime* as ineffective as *prime*.

Another idea is to base primality tests on the observation that only for a prime p it holds that for any $0 < k < p$:

$$\binom{p}{k} \equiv 0 \pmod{p}. \tag{1.65}$$

But, again, we have to test this for all ks. In the case of 561, for instance, 446 of the 560 possible ks fulfil the equation. But computing all binomial coefficients $\binom{p}{1}, \binom{p}{2}, \ldots \binom{p}{p-1}$, is not feasible for large numbers.

The next bunch of ideas would use an implication of that fact, such as *freshman's dream*:

$$(a+b)^p \equiv a^p + b^p \pmod{p} \tag{1.66}$$

Unfortunately, there is a proof (by Ghatage and Scott) that this condition is true exactly if p is prime or ... a Carmichael number. To distinguish primes from Carmichael numbers, it is again necessary to test for all numbers a.

There is finally an efficient test that builds on a variant of freshman's dream, namely the Agrawal-Kayal-Saxena (AKS) test, which exploits the fact that

$$(x-a)^p \equiv (x^p - a) \pmod{p}. \tag{1.67}$$

Agrawal, Kayal and Saxena, a group of Indian mathematicians and computer scientists won the Gödel Prize for their paper "PRIME is in P" where they actually presented the AKS test. The test adopts algebraic methods to avoid computing all possible xs and as to establish that a number is prime. Since we have not looked into algebra yet, we have to postpone the discussion of this algorithm. In practical terms, this is not an issue, since there are algorithms that establish the primality of a number with sufficiently high probability.

It is a very common approach in math and computer science to accept algorithms with bounded error probability if those algorithms are significantly faster or simpler than their deterministic cousins and if an error bound can be given. This bound can then be

used to compute the number of repetitions necessary to make the probability of failure small enough to be ignored.

A test able to establish the primality of a number greater than 2 with sufficient probability is the Rabin-Miller test. The mathematical idea is based on Fermat's little theorem, but with some refinement. The reasoning starts with the observation that $p-1$ in a^{p-1} in Fermat's equation is even, since p is prime, as required by the theorem, and $p > 2$, as required by Rabin-Miller. We could therefore represent that number as a series of squares of the form

$$a^{s^{2^{2^{\cdots}}}},$$

for some odd integer s. This is of course equivalent to

$$a^{s\times 2\times 2\times \cdots}.$$

If p is, say, 5, then the Fermat equation would look like $a^4 \equiv \pmod 5$, which we could write as $a^{1\times 2\times 2}$, where $s = 1$. For $p = 7$, this would look like $a^{3\times 2}$ and $s = 3$.

Let us look at the "last" square $(a^d)^2$ independent of whether d is even or odd. We know this square must make the equation congruent to 1 modulo p. Let us examine this last exit before $p-1$, a^d, and call this number x. We then have:

$$x^2 \equiv 1 \pmod p,$$

From here, we can apply the same technique we have already used to prove Wilson's theorem; we first subtract 1 on both sides and we get

$$x^2 - 1 \equiv 0 \pmod p.$$

$x^2 - 1$ can be factored into $(x+1)(x-1)$ and we get the congruence

$$(x+1)(x-1) \equiv 0 \pmod p.$$

For the product on the left-hand side to become 0, one of its factors must be 0. For the first to be 0, x must equal -1; for the second, x must equal 1. This is the same result we have already obtained, when proving Wilson's theorem.

We substitute a^d back for x and see that the last exit before the last square must be either $a^d \equiv -1 \pmod p$ or $a^d \equiv 1 \pmod p$.

The second case, however, where a^d is 1, can only occur (for p prime) if a^s was 1 or $p-1$ right from the beginning or, if somewhere on the way from a^s to a^d, the whole expression

became $p-1$. We know this for sure from Wilson's theorem: For every number a in the range $1 \ldots p-1$, there is an inverse a', such that $aa' \equiv 1 \pmod{p}$ and there are only two numbers for which $a = a'$, namely 1 and $p-1$. Squaring a number that is neither 1 nor $p-1$, therefore, cannot result in 1 (if p is prime).

There are thus only two ways for a^{2d} to become 1: either a^s was 1 right from the beginning, then squaring will not change anything; or, at some point, a^d is $p-1$ (which may be obtained by squaring two numbers) and then, in the next step, 1. It is impossible for 1 to pop up on the way without $p-1$ occurring before – if p is prime.

This is the idea of Rabin-Miller: It finds an odd number s and a number t that tells us how often we have to square a^s to get to a^{p-1}. Then it checks if a^s is either 1 or $p-1$. If not, it checks if any of a^s, a^{2s}, a^{4s}, …, a^{ts} is $p-1$. If this is not the case, p is composite.

The advantage of this method over the simple Fermat test is that it reduces the set of remainders per number that actually pass. This also reduces the probability of the test to actually go wrong for a specific number. If this probability is significantly less than 50%, we can reach a correct result with high probability by applying the test more than once. But let us postpone the probability reasoning for a short while. First, we will have a look at the implementation of the algorithm.

To start, we need a function that gives us s, the odd number after taking all squares out of $p-1$, and t, the number that tells us how many squares we have actually taken out to reach s. It then holds that $2^t s = p-1$. In the lack of a useful name for that function, we call it *odd2t*:

```
odd2t :: Natural → Natural → (Natural, Natural)
odd2t s t | even s     = odd2t (s `div` 2) (t + 1)
          | otherwise = (s, t)
```

For 16, *odd2t* would give $(1, 4)$, since 16 is a power of 2, *i.e.* divided subsequently by 2, it will reach 1. One has to multiply 1 4 times by 2 to get 16 back: $1 \times 2^4 = 16$. For 18, *odd2t*, accordingly, would yield $(9, 1)$, since $9 \times 2^1 = 18$.

The next function is the primality test itself:

```
rmPrime :: Natural → Natural → Natural → Natural → Bool
rmPrime p a s t = case (a ↑ s) `rem` p of
                    1 → True
                    v → if v ≡ p − 1 then True
                                     else go t v
  where go 1 _ = False
        go t v = case (v ↑ 2) `rem` p of
                   1 → False
                   v' → if v' ≡ p − 1 then True
                                      else go (t − 1) v'
```

The function receives four arguments: The number to test for primality, the test can-

didate a, also called a witness for the primality of p, and s and t obtained from *odd2t*. The function raises a to the power of s. If the result is 1 or $p-1$ modulo p, p has already passed the test. Otherwise, we loop through *go*. *go* receives two argument t and v (which initially is a^s). If t is 1, we have exhausted all the squares in $p-1$ without having seen $p-1$. The test has failed. Otherwise, we create the next square modulo p. If this square is 1, something is wrong: we know that v was neither 1 nor $p-1$, so squaring it cannot result in 1, if p is prime, because only 1 and $p-1$ are their own inverses. Otherwise, if the result is $p-1$, we are done. Further squaring will yield 1 and all conditions for p being a prime are fulfilled. Otherwise, we continue with the next square, reducing the square counter t by 1.

The function that brings these bits together needs randomness to choose as. To this end, we use the function *randomNatural* defined in the previous chapter:

```
rabinMiller :: Natural → Natural → IO Bool
rabinMiller _ 0 = return False
rabinMiller _ 1 = return False
rabinMiller _ 2 = return True
rabinMiller k p | even p     = return False
                | otherwise = let (s, t) = odd2t (p − 1) 0 in go k s t
  where go 0 _ _ = return True
        go i s t  = do a ← randomNatural (2, p − 1)
                       if rmPrime p a s t then go (i − 1) s t
                                          else return False
```

The function receives two arguments: k and p. p is the number under test. k tells the function how often it has to repeat the test until the expected probability is reached. If k is exhausted, *i.e.* $k = 0$, we return *True* (in *go*) and p has passed the complete test.

At the beginning, we take care of some trivial cases, such as 0 and 1, which are never prime, and 2, which actually is prime. With the exception 2, no even number is prime. Then we start the hard work: we first find s and t using *odd2t*; then we enter *go*. We generate an a from the range $2 \ldots p-1$, using *randomNatural*. Then we apply the test. If the test fails, we immediately return False. If the test passes, we repeat until $i = 0$.

To reason about the probability for the test to fail, let us look at some examples. The following simple function can be applied to a number to show the results of the Fermat test. It returns a list of tuples where the first element is one of the numbers $2 \ldots n-1$ and the second is this number raised to $n-1$ mod n:

```
rest :: Natural → [(Natural, Natural)]
rest n = zip rs $ map (λa → (a ↑ (n − 1)) `rem` n) rs
  where rs = [2 .. n]
```

rest 9, for instance, yields:

$[(2,4),(3,0),(4,7),(5,7),(6,0),(7,4),(8,1)]$

We see that, for most of the numbers, the Fermat test would fail. For 8, however, it would pass, since $8^8 \equiv 1 \pmod 9$. 8 is therefore a *liar* concerning the primality (or, more precisely, for the compositeness) of 9. Unfortunately, Rabin-Miller would not help us in this case, since *odd2t* 8 0 = (1, 3); 8^1, however, is $n - 1$ and the test would immediately pass. 8, hence, is a *strong liar* for 9.

Let us look at another example: 15. *odd2t* for 15 gives (7, 1), since 14 ‘*div*‘ 2 is 7, which is odd. *rest* 15 yields:

$[(2,4),(3,9),(4,1),(5,10),(6,6),(7,4),(8,4),(9,6),(10,10),(11,1),(12,9),(13,4),(14,1)]$.

There are several liars: 4, 11 and 14. 14, again is a strong liar, since $14^7 \bmod 15 = 14$, which is $n - 1$. 11 and 4, however, are ruled out by Rabin-Miller: $11^7 \bmod 15 = 11$, which squared would never be 1, if 15 were prime; $4^7 \bmod 15 = 4$, which squared, again, would not result in 1, if 15 were prime.

Let us devise a function that counts the occurrences of (Fermat) liars and strong liars for any given composite n. The first function is called *liars* and quite simple:

```
liars :: Natural → Int
liars = length ∘ filter (≡ 1) ∘ map snd ∘ rest
```

That is we start with *rest*, ignore the first element of each tuple, filter the 1s and count the elements of the resulting list. (The return type of the function is *Int*, rather than *Natural*, because we use *length*, which returns an *Int* anyway.)

The strong liar function is a bit more tricky:

```
strongLiars :: Natural → Int
strongLiars n | even n     = 0
              | otherwise =
  let (s, t) = odd2t (n − 1) 0
      sl     = foldr (λa l → detector l a s t) [] [2 .. n − 1]
  in  length sl
  where detector l a s t | rmPrime n a s t = a : l
                        | otherwise       = l
```

For finding strong liars, we implement a part of the Rabin-Miller test and, therefore, we ignore even numbers (just yielding 0). For even numbers, the (s, t) values would not make any sense, since $n - 1$ is odd! Then, we apply the *rmPrime* test we implemented for Rabin-Miller to all remainders, adding those that pass to the result list. Finally, we just yield the length of that list.

If we apply *liars* to a prime number p, all witnesses $2 \ldots p - 1$ are counted as liars, *e.g.* *liars* 11: 9. The same is true for *strongLiars*, since the fact that all witnesses are strong liars could be a definition of primalitiy. Thus, *strongLiars* 11: 9.

Applied to 9, *liars* and *strongLiars* yield 1. Applied to 15, *liars* yields 3; *strongLiars*, however, yields only 1. This is in-line with our investigation above. Here are some more examples for numbers between 21 and 75:

21	25	27	33	35	39	45	49	51	55	57	63	65	69	75
3,1	3,3	1,1	3,1	3,1	3,1	7,1	5,5	3,1	3,1	3,1	3,1	15,5	3,1	3,1

We see a quite colourful picture. Many numbers have 3 liars and 1 strong liar; for some numbers, there is no difference in liars and strong liars, for instance 25 and 27, both have the same numbers of liars and strong liars, namely 3 and 1. Other numbers, *e.g.* 45, show a strong reduction in going from liars to strong liars. There are some peaks, *e.g.* 65 has 15 liars and 5 strong liars, much more liars than most other numbers. If we continue up to 99, the greatest number we will see is $(35, 17)$ for 91. The nasty number 341 has 99 liars and 49 strong liars. Finally, here are the dreadful Carmichael numbers:

561	1105	1729	2465	2821	6601	8911
319,9	767,29	1295,161	1791,69	2159,269	5279,329	7127,1781

For many Carmichael numbers, the reduction of liars is significant – for some, the reduction is about factor 10 – 30. There are some exceptions with reduction of a factor of "only" 6 like 8911. In general, the number of strong liars is very low compared to n, the prime candidate. It can be shown in fact that the number of strong liars for an odd composite n is at most $\frac{n}{4}$. (This has actually been shown with contributions, among others, by the legendary Paul Erdős, 1913 – 1996.) The arguments, however, are much beyond our scope.

The ratio $\frac{n}{4}$ implies that the probability of hitting a strong liar, when performing *rmPrime* on a randomly chosen witness for n, is $\frac{1}{4}$. In other words, one has to try four times in average to get a strong liar by chance. When we repeat the test several times, we reduce the probability that **all** witnesses we have used are strong liars. It is important to notice that the test yields *False* immediately, when we find a witness for compositeness. We continue only if all tests so far have been witnesses for primality.

The probability is therefore computed as $\frac{1}{4^k}$, where k is the number of repetitions. The probability to obtain two liars in two applications is $\frac{1}{4^2} = \frac{1}{16}$. We, hence, would have to call a Rabin-Miller Test with two repetitions 16 times in average to test only on strong liars once. With four repetitions, the denominator is $4^4 = 64$, with eight, it is 65 536, with sixteen, it is 4 294 967 296 and so on. A reasonable value for k to defend against malicious attacks given, for example, in *Cryptographic Engineering* is $k = 64$. In average, one has a chance of 1 out of 4^{64} or 340 282 366 920 938 463 463 374 607 431 768 211 456 to hit only strong liars in testing a number for primality. That, indeed, appears to be reasonable. A final version of the Rabin-Miller Test could then look like this:

```
rmptest :: Natural → IO Bool
rmptest 0 = return False
rmptest 1 = return False
rmptest 2 = return True
rmptest n | even n     = return False
          | otherwise = rabinMiller 64 n
```

1.11 Primes in Cryptography

To say this right at the beginning: this is not an introduction to cryptography! We will go through some examples of how primes are used in cryptography, in particular the Diffie-Hellman key exchange protocol and RSA. But we will not discuss pitfalls, common errors or other issues you have to take care of when implementing cryptosystems. If you want to learn about cryptography, you definitely have to study the literature on the topic. Cryptography poses very hard engineering challenges and we are far from adressing them here. Again: this is not an introduction to cryptography! We do not even use a big number library that would be able to cope with numbers with hundreds or thousands of digits. When you use the algorithms presented here (like the Rabin-Miller test) on numbers of that size, you will probably have to wait minutes or even hours for results. Raising a number of thousands of digits to another number of thousands of digits requests special handling. You cannot do that with our *Natural* data type or even the Haskell *Integer* data type. So, once again: this is not an introduction to cryptography.

Primes come into play in cryptography, typically, in very special applications. Usually, to encrypt and decrypt a message, secret keys are used that are known to both sides of the communication, traditionally called Alice and Bob in the literature. The algorithms to convert the plain message into the *cyphertext* have in most cases nothing to do with primes, but are rather combinations of basic operations like XOR and *bit shuffle* involving the original message and the key material.

The weakest link in this kind of cryptography is the key itself. All parties that take part in the secure communication must know the key and, in consequence, the key must be shared among them in a secure way. How to share the key safely is indeed a major challenge. For the German submarines in World War II, just to name a popular example, the loss of a codebook was at least as challenging as the cryptoanalysis by British specialists at Bletchley Park. A new codebook could hardly be distributed among all submarines on the ocean in due time.

The challenge is an instance of the bootstrapping problem: we have to start a process, namely secure communication, without having the means to run this process, namely secret keys distributed among all parties. One solution is to use publicly available information to convert a plain message into a cyphertext by means of a *one-way function*, *i.e.* a function that is easy to calculate, but difficult to revert. Here is where primes come in.

A one-way function f, is a function that computes a result from a given input, such as $f(x) = y$, for which no inverse f' is known, such that $f'(y) = x$ or $f'(f(x)) = x$. Plain multiplication, for instance, is not a good one-way function, since with the result y known, we simply can revert the effect by division. If we have a function $f(x) = ax$, and a cyphertext $f(x) = y$. We can reconstruct x, simply, by $f'(y) = y/a$. Multiplication would be a good one-way function, however, if both a and x were unknown. The inverse, would then be factoring – and factoring of large numbers is indeed a hard problem.

Before we have a closer look at concrete examples, we have to come back to a question we have already discussed (but without satisfying results), namely how to generate huge primes. We have seen Mersenne primes and Fermat primes, but little is known about this kind of numbers and in particular, no recipe is known how to find new ones. Practial algorithms are in fact much simpler in terms of mathematical ideas. Prime generators usually generate a random number in a given range, test whether it is prime and, if it is, return this number or, if it is not, try again.

A reasonable prime generator for natural numbers, using once again the random number generator *randomNatural*, could look like this:

$$
\begin{array}{l}
\mathit{generatePrime} :: \mathit{Natural} \to \mathit{IO}\ \mathit{Natural} \\
\mathit{generatePrime}\ k = \mathbf{do}\ n \leftarrow \mathit{randomNatural}\ (2 \uparrow (k-1), 2 \uparrow k - 1) \\
\qquad\qquad\qquad\qquad\quad t \leftarrow \mathit{rmptest}\ n \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ t\ \ \mathbf{then}\ \mathit{return}\ n \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{generatePrime}\ k
\end{array}
$$

As you can see, this is just a trial-and-error approach using the Rabin-Miller test to check whether a given number is prime. Is there not a risk of running a lot of time depending on the range we have chosen? Indeed, it can take a lot of time, before we actually find a prime. However, we know some things about the distribution of primes, so that we are guaranteed to find a prime within a reasonably chosen range. We will discuss some of the relevant aspects concerning the distribution of primes in the next section. For the moment, it may suffice to mention that a range like $k \ldots k + c$, where c is some constant smaller than k, is obviously not sufficient. The range above, however, is chosen in terms of powers of 2 and, assuming that k is at least 10, we are guaranteed to find some prime numbers in such a range like, for $k = 10$, $512 \ldots 1023$. It is quite common, by the way, to give the size of the prime wanted in bits, *i.e.* in terms of the number of digits in binary representation, rather than in decimal representation.

The first example of prime-based key exchange is the Diffie-Hellman protocol developed by Whitfield Diffie, Martin Hellman and Ralph Merkle in the 70ies. There was another group of scientists that developed a similar algorithm shortly before Diffie and Hellman published their results. Those scientists, unfortunately, were working for the British secret service, GCHQ, at the time and were not allowed to publish their results. They, thus, escaped eternity.

The Diffie-Hellman protocol is very simple and elegant, but has some pitfalls that must

be addressed. Its main purpose is to exchange a secret key between Alice and Bob without Eve, another fictional character famous in cryptography literature, getting to know that key by eavesdropping. Before the Diffie-Hellman protocol actually starts, Alice and Bob have agreed somehow on a public key that may be known to anyone including Eve. This public key may be assigned to either Alice or Bob and may be registered in a kind of phone book or it may be agreed upon in the *handshaking* phase of the protocol.

The public key consists of two parts: a prime number p and a generator g. When Alice initiates the protocol, she chooses a number x from the range $2 \ldots p-2$, computes $g^x \bmod p$ and sends this number to Bob. Bob, in his turn, chooses a number y from the same range, computes $g^y \bmod p$ and sends the result back to Alice. Evil Eve knows p and g and may see g^x and g^y. But none of these values is actually the key. The key, instead, is g^{xy}. At the end of the protocol, this number is known to Alice and Bob. Each of them just has to raise the number he or she receives from the other by his or her own number. So Alice chooses x and sends g^x to Bob. Bob chooses y and sends g^y. Alice computes the key as $k = g^{y^x}$, *i.e.* she raises the number she receives from Bob to her own number; Bob computes the key, accordingly, as g^{x^y}, *i.e.* he raises the number he receives from Alice to his own number. That is all.

For a simple example, let us assume the public key is $p = 11$ and $g = 6$. Now, Alice chooses a random number from the range $2 \ldots 9$, say, 4, and Bob likewise, say, 3. Alice computes $6^4 \bmod 11 = 9$ and sends 9 to Bob. Bob computes $6^3 \bmod 11 = 7$ and sends 7 to Alice. Alice computes the key as $k = 7^4 \bmod 11 = 3$ and Bob computes it as $k = 9^3 \bmod 11 = 3$. The result is the same for both, of course, because, eventually, both have performed the same operation: g^{xy}.

The security is based on the difficulty to solve the equation

$$k = g^{xy}, \tag{1.68}$$

where g, g^x and g^y are known. This is an instance of the *discrete logarithm* problem, the logarithm in a finite field. If k, g and xy were ordinary real numbers, we could solve the equation in three steps: $x = log_g(g^x)$, $y = log_g(g^y)$ and, finally, $k = g^{xy}$. For the discrete logarithm, however, no efficient solution is known today.

In a trivial examples like the one we used above, Eve can simply try out all possible combinations. In real cryptography applications, we therefore have to use very large primes. But the security of the algorithm also depends on the order of g. For instance, if we want a security that corresponds to 1000 bits (a number with more than 300 digits in decimal representation), then the order of g should be much more than some hundreds or thousands or even millions of numbers generated by that g. To choose a proper g is therefore essential for the strength of the protocol. The question now is: how to choose a proper g?

If you have carefully read the previous sections, you already know the answer: we must use a g from the Schnorr group of a safe prime. Indeed, knowing the order of g is again a hard problem. We have to solve the equation $g^k = 1 \mod p$, which is again an instance of the discrete logarithm. We can circumvent this problem of finding an appropriate group by finding an appropriate prime. With a safe prime, the selection of an appropriate g is indeed simple. We just have to avoid one of the trivial groups, that is we have to avoid 1 and $p-1$, then we are guaranteed that g is in the group of the Sophie Germain prime q or in the larger group of the safe prime $2q+1$.

Sophie Germain primes, hence, give us a means to reduce the difficult problem of finding a proper g to the much simpler problem of selecting a proper prime:

```
safePrime :: Natural → IO Natural
safePrime k = generatePrime k >>= λq → let p = 2 * q + 1 in do
  t ← rmptest p
  if t then return p
       else safePrime k
```

This function generates a *safe prime*, *i.e.* a prime of the form $2q+1$, where q is a Sophie Germain prime. It just generates a random prime q, doubles it and adds 1 and checks whether the resulting number p is again prime and, if not, repeats the process. From the group of this prime, we can then take a random g, $g \neq 1, g \neq p-1$, and this g belongs in the worst case to the group q of order $\frac{p-1}{2}$. Since q was generated according to security level k, the group of q is exactly what we need and our main issue is solved.

There is still a problem, though. From the communication between Alice and Bob, Eve sees g^x and g^y. If she has read the section on quadratic residues, she can determine whether g is a square (modulo p) using the Legendre symbol. If g is a nonresidue, then she has an attack: she can repeat the test on the numbers she sees, namely, g^x and g^y. If a number of the form g^z is a nonresidue, where g as well is a nonresidue, then z is odd, otherwise, z is even. This is because even exponents are just repeated squares. So, if z is even, g^z is a residue and, otherwise, it is not.

We should avoid this leakage. Even though Eve does not learn the whole number, she gets an information she is not entitled to have. The leakage effectively reduces the security level by the factor 2, since Eve learns the least significant digit of the number in binary representation: odd numbers in binary representation end on 1, even numbers on 0. We can avoid this problem simply by chosing a residue right from the beginning. The function to generate an appropriate g could then look like this:

```
generator :: Natural → IO Natural
generator p = do a ← randomNatural (2, p − 2)
                 let g = (a ↑ 2) `rem` p
                 if g ≡ p − 1 then generator p
                              else return g
```

We select an a from the range $2 \ldots p-2$ and square it. The number, hence, is guaranteed

to be a residue eliminating the problem discussed above. We test if $g = p - 1$, which is forbidden, since it is in a trivial group. a^2, as you know, can become $p - 1$, if $p \equiv 1 \pmod 4$. You may argue that this is not the case, when p is a safe prime, because, if $\frac{p-1}{2}$ is prime, then $p - 1$ cannot be a multiple of 4. It is a good practice, however, to keep different components of the security infrastructure independent of each other. In spite of the fact that we usually use *generator* with safe primes, it could happen that someone uses it with another kind of prime (for example a prime of the form $nq + 1$). This test simply avoids that anything bad happens under such circumstances.

Note that we really do not need to check for $g = 1$, since a^2, with a chosen from the range $2 \ldots p - 2$, excluding both 1 and $p - 1$, can never be 1.

The next function initialises the protocol. It is assumed that p and g are known already to all involved parties when it is called:

```
initProtocol :: Chan Natural → Chan Natural →
                Natural → Natural → IO Natural
initProtocol inch outch p g = do
  x  ← randomNatural (2, p − 2)
  let gx = (g ↑ x) `rem` p
  writeChan outch gx
  gy ← readChan inch
  unless (checkG p gy) $ error ("suspicious value: " ++ show gy)
  return ((gy ↑ x) `rem` p)
```

The function receives four arguments: An input channel and an output channel and p and g. It starts by generating a random x from the range $2 \ldots p - 2$. It then computes $g^x \bmod p$ and sends the result through the outgoing channel. Then it waits for an answer through the incoming channel. When a response is received, the value is checked by *checkG*, at which we will look in an instant. If the value is accepted, the function returns this value raised to x modulo p. This is the secret key.

It is actually necessary to check incoming values, since Eve may have intercepted the communication and may have sent a value that is not in the Schnorr group. To protect against this *man-in-the-middle* attack, both sides apply the following tests:

```
checkG :: Natural → Natural → Bool
checkG p x = x ≢ 1 ∧
             x < p ∧
             legendre (fromIntegral x)
                      (fromIntegral p) ≡ 1
```

For any value x received through the channel, it must hold that $x \neq 1$ (because 1 is in the wrong group), it must also hold that $x < p$, *i.e.* it must be a value modulo p, and x must be a residue of p. This is because we have chosen g to be a square and any square raised to some power is still a residue of p. So, if one of these conditions does not hold,

something is wrong and we immediately abort the protocol.

Now, we look at the other side of the communication:

```
acceptProtocol :: Chan Natural → Chan Natural →
                  Natural → Natural → IO Natural
acceptProtocol inch outch p g = do
  gx ← readChan inch
  unless (checkG p gx) $ error ("suspicious value: " ++ show gx)
  y  ← randomNatural (2, p − 2)
  let gy = (g ↑ y) `rem` p
  writeChan outch gy
  return ((gx ↑ y) `rem` p)
```

The function starts by waiting on input. When it receives some input, it checks it using *checkG*. If the value is accepted, the function generates a random y from the range $2 \ldots p-2$, computes $g^y \bmod p$ and sends it back; finally the key is returned.

Now, the protocol has terminated, both sides, Alice and Bob, know the key and they can start to use this key to encrypt the messages exchanged on the channel. Since, as mentioned several times, this is not an introduction to cryptography, we have omitted a lot of details. One example is a robust defence against *denial-of-service* attacks. In the code above, we wait for input forever. This is never a good idea. An attacker could initiate one protocol after the other without terminating any of them. The server waiting for requests would quickly run out of resources.

A cryptosystem with somewhat different purposes than Diffie-Hellman is RSA, named after its inventors Ronald Rivest, Adi Shamir and Leonard Adleman and published in 1978. Similar to Diffie-Hellman, RSA aims to provide an encryption system for key exchange, but, beyond encryption, RSA is also designed as an authentication system based on electronic signatures. The latter implies that the public key is assigned to a person. Furthermore, an infrastructure is needed that inspires the confidence that a given public key is really the key of the person one believes one is communicating with. There are many ways to create such an infrastructure. It may be community-based, for example, so that people one trusts guarantee for people they trust; another way is that trusted organisations provide phone books where signatures can be looked up. All possible implementations depend at some point on trust. In fact, trust is just the other side of the security coin: without trust, there is no security.

When Alice uses RSA to send a secret message to Bob, she uses Bob's public key to encrypt the message, and Bob, later, uses his private key to decrypt the message. The design of RSA guarantees that it is extremely difficult to decrypt the cyphertext without knowledge of Bob's private key. If Alice, additionally, wants to sign the message to make sure that Eve cannot exchange her messages by other ones, she uses her own private key to create a signature on the message (usually by first creating a *hash* of that message). Bob can then assure himself that the message was really sent by Alice by verifying the

signature using Alice's public key. The role of public and private key, hence, is swapped in encryption and signing. Encryption is done with the addressee's pupblic key and undone by the addresse's private key; signing is done with the sender's private key and approved with the sender's public key.

Just as Diffie-Hellman, RSA is based on modular arithmetic – but with a composite modulus. With a composite modulus, some care must be taken, of course, since it does behave as a prime only with respect to those numbers that are coprime to it. The modulus n is created by multiplying two large primes, p and q. This guarantees that n behaves like an "ordinary" prime with respect to most numbers in the range $1 \ldots n-1$, *viz.* all numbers that are not multiples of p and q.

We also need a number t, such that for numbers a coprime to n, it holds that $a^t \equiv 1 \pmod{n}$. For a prime number p, as we know from Fermat's little theorem, this t would be $p-1$. For the prime factors of n, hence, $p-1$ and $q-1$ would do the trick. But any multiple of $p-1$ and $q-1$ would do the trick as well, including of course $(p-1)(q-1)$ or $\mathrm{lcm}(p-1, q-1)$, the least common multiple of $p-1$ and $q-1$.

Now, consider Fermat's theorem written like this:

$$a^p \equiv a \pmod{p} \tag{1.69}$$

and try to solve the following congruence system:

$$\begin{aligned} x &\equiv a \pmod{p} \\ x &\equiv a \pmod{q}. \end{aligned}$$

An obvious solution to this system, according to the Chinese Remainder theorem, is the number x that is congruent to a modulo pq. Let us write x as a^t, where $t = \mathrm{lcm}(p-1, q-1)$:

$$\begin{aligned} a^t &\equiv a \pmod{p} \\ a^t &\equiv a \pmod{q}. \end{aligned}$$

An obvious solution to this system, still according to the Chinese Remainder theorem, is $a^t \equiv a \pmod{pq}$. Let us look at an example: $p = 7$, $q = 11$ and $n = pq = 77$. The lcm of $p-1 = 6$ and $q-1 = 10$ is 30. Therefore, for any number a: $a^{30} \equiv 1 \pmod{7}$ and $a^{30} \equiv 1 \pmod{11}$, but also: $a^{31} \equiv a \pmod{7}$ and $a^{31} \equiv a \pmod{11}$ and, this is important, $a^{30} \equiv 1 \pmod{77}$ and $a^{31} \equiv a \pmod{77}$. For instance $a = 3$: $3^{30} \equiv 1 \pmod{7}$, $3^{31} \equiv 3 \pmod{7}$ and $3^{30} \equiv 1 \pmod{11}$ and $3^{31} \equiv 3 \pmod{11}$. But also: $3^{30} \equiv 1 \pmod{77}$ and $3^{31} \equiv 3 \pmod{77}$.

Note that this is nothing new. We just applied the Chinese Remainder theorem to a quite trivial case. However, from this trivial case (and with some help from Euler as we will see later), an important theorem follows, namely *Carmichael's theorem* that can be stated in the scope of our problem here as: the least number t fulfilling the congruence $a^t \equiv 1 \pmod{n}$ for any number a coprime to n is the lcm of $p_1 - 1, p_2 - 1, \ldots p_s - 1$, where $p_1 \ldots p_s$ are the prime factors of n. Since our number n has only two prime factors, namely p and q, our t is $\text{lcm}(p-1, q-1)$. So, finally, this fellow Carmichael is not only bugging us with crazy numbers, but he actually lends a hand to solve a problem once in a while! Thanks Robert!

The importance for the RSA system is related to the fact that we need a public number to compute the cyphertext and a private number to reconstruct the original message. The method to compute the cyphertext is exponentiation. For this purpose, we need an exponent called e. The number e is chosen such that it is a small odd number, $1 < e < t$ and e does not divide t nor n. Now we find the inverse e' of e mod t, such that $ee' \equiv 1 \pmod{t}$.

When Alice encrypts a message m, she computes $c = m^e \mod n$. To decrypt the cyphertext c, Bob computes $c^{e'}$, which is $m = m^{e^{e'}} = m^{ee'} \mod n$. Modulo t $m^{ee'}$ would just be $m^1 = m$, since $ee' \equiv 1 \pmod{t}$. Modulo n, this number is some multiple of t plus 1: $ee' \equiv kt + 1 \pmod{n}$. We, hence, have $m^{kt+1} = m^{kt} \times m = m^{t^k} \times m$. But since $m^t \equiv 1 \pmod{n}$, due to the Carmichael theorem, we have $1^k \times m = 1 \times m = m$. Again, Mr Carmichael, thank you!

To resume what we need for RSA: We have a public key (n, e) and a private key (p, q, t, e'), where $n = pq$, $t = \text{lcm}((p-1), (q-1))$ and $ee' \equiv 1 \pmod{t}$. It is essential that all components of the private key remain secret. Any component that leaks out helps Eve reveal the entire private key. The core of the secret is e', since it can be used directly to decrypt encrypted messages and to sign messages in the name of its owner. With t revealed, e' can be simply computed with the extended gcd algorithm; with one of p or q revealed, the respective other factor of n can be simply computed by $q = n/p$ or $p = n/q$. With p and q both known, however, t can be computed by means of the lcm. This boils down to the fact that the security of RSA depends on the difficulty of the discrete logarithm and the factoring of large numbers.

Since, in any concrete implementation of RSA, we have to refer to the components of the keys quite often, let us define data types to encapsulate the public and the private information:

```
type PublicK = (Natural, Natural)
type PrivateK = (Natural, Natural, Natural, Natural)

pubN, pubE :: PublicK → Natural
pubN = fst
pubE = snd

privP, privQ, privT, privD :: PrivateK → Natural
privP (p, _, _, _) = p
privQ (_, q, _, _) = q
privT (_, _, t, _) = t
privD (_, _, _, d) = d
```

This is just two type synonyms for private and public key and a set of accessor functions. Note that we call the inverse of e in the private key d as it is often referred to in the literature.

As for Diffie-Hellman, chosing good values for public and private key is an essential part of the system. The following function is a reasonable key generator:

```
generateKeys :: Natural → IO (PublicK, PrivateK)
generateKeys k = do p      ← genPrime 1
                    q      ← genPrime p
                    let t = lcm (p − 1) (q − 1)
                    pair   ← findED 1000 t
                    case pair of
                         Nothing    → generateKeys k
                         Just (e, d) → if e ≡ d ∨
                                          e ≡ p ∨ e ≡ q ∨
                                          d ≡ p ∨ d ≡ q
                                       then generateKeys k
                                       else let pub = (p ∗ q, e)
                                                priv = (p, q, t, d)
                                            in return (pub, priv)
  where genPrime p = do q ← generatePrime (k `div` 2)
                        if p ≡ q then genPrime p
                                 else return q
```

We start by generating two primes using *genPrime*. *genPrime* incorporates a test to ensure that we do not accidently choose the same prime twice. When we call *genPrime* for p, we pass a number that is certainly not equal to the generated prime, namely one. In the second call to generate q, we pass p. Then we create t as $\text{lcm}(p-1, q-1)$ and then we find the pair (e, d) using *findED*, at which we will look next. *findED* returns a *Maybe* value. If the result is *Nothing*, we start all over again. Otherwise, we ensure that e and d differ and that e and d differ from p and q. This is to ensure that we do not accidently publish one of the secrets, namely e or one of the prime factors of n. Finally, we create the public key as (pq, e) and the private key as (p, q, t, d).

The most intriguing part of the key generator is finding the pair (e, d). Since t is not a prime number, we are not guaranteed to find an inverse for any $e < t$. So, we may need several tries to find an e. But it might even be that there is no such pair at all for t. Since t depends on the primes p and q, in such a case, we have to start from the beginning. As an example, consider the primes $p = 5$ and $q = 7$; t in this case is 12. If we try all combinations of the numbers $2 \ldots 10$, we see that the only pairs of numbers whose product is 1 are 5×5 and 7×7. In this case: $e = d$, an option that any attacker will probably try first and should therefore be avoided. Here is an implementation of *findED*:

```
findED :: Natural → Natural → IO (Maybe (Natural, Natural))
findED 0 _ = return Nothing
findED i t = randomNatural (7, t − 2) >>= λx →
  let z |  even x        = x − 1
        |  otherwise     = x
    (e, d)               = tryXgcd z t
  in if (e ∗ d) `rem` t ≡ 1 then if e > t `div` 2 then return (Just (d, e))
                                                  else return (Just (e, d))
                            else findED (i − 1) t
  where tryXgcd a t = case nxgcd a t of
                        (1, k) → (a, k)
                        _      → (0, 0)
```

The function takes two arguments. The first is a counter that, when expired, indicates that we give up the search with the given t. As you can see above in *generateKeys*, the counter is defined as 1000. That is just some randomly chosen value – there may be better ones. We could try to invent some ratio for t such as half of the odd numbers smaller than t. But t is a very large number. Any ratio would force us to test single numbers for hours if we are unlucky.

We then choose a random number from the range $7 \ldots t - 2$ as a candidate for e. We do not want $e = t - 1$, since the inverse in this case is likely to be e itself. We start with 7, since we want to avoid very small values for e. We also want e to be odd. Otherwise, it will not be coprime to t, which is even.

We then compute (e, d) by means of *tryXgcd*. This function computes the *gcd* and the inverse by means of *nxgcd*, which we defined in the section on modular arithmetic. If the *gcd* is not 1, then we are unlucky, since e and t must be coprime. In this case, we return $(0, 0)$, a pair that certainly will not pass the test $ed \equiv 1 \pmod{t}$ and, this way, we cause the next try of *findED*. Otherwise, we return the pair (a, k), e and its inverse modulo t.

If we have found a suitable pair in *findED*, we return this pair either as (e, d) or as (d, e), if $e > t$`div`2. The reasoning is that encryption will be more efficient with a small e. On the other hand, we do not want to impose any explicit property on d (such as $d > e$), since that would be a hint that reduces the security level.

The key generator is usually not used, when we establish a secure communication. Instead, the key pair is considered to be stable as long as it has not been compromised by loss or by an attack on the server where key pairs are stored. Some stability is necessary for the authentication part of RSA. If public keys changed frequently, it would be more difficult to be sure that a given key really belongs to the person one thinks it belongs to.

Once the keys are available and the public key has been published, Alice can encrypt a message to Bob using the encryption function:

$$\begin{aligned}&\mathit{encrypt} :: \mathit{PublicK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{encrypt}\ \mathit{pub}\ m = (m \uparrow (\mathit{pubE}\ \mathit{pub}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

Alice uses this function with Bob's public key and a message m represented as a (large) integer value. The function raises the message m to e and takes the result modular n. Bob can decrypt the cyphertext using his private key:

$$\begin{aligned}&\mathit{decrypt} :: \mathit{PublicK} \to \mathit{PrivateK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{decrypt}\ \mathit{pub}\ \mathit{priv}\ c = (c \uparrow (\mathit{privD}\ \mathit{priv}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

To sign a message, Alice would use her private key:

$$\begin{aligned}&\mathit{sign} :: \mathit{PublicK} \to \mathit{PrivateK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{sign}\ \mathit{pub}\ \mathit{priv}\ m = (m \uparrow (\mathit{privD}\ \mathit{priv}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

Signing, hence, is just the same as decryption, but on a message that was not yet encrypted. To verify the signature, Bob uses Alice's public key with a function similar to encryption:

$$\begin{aligned}&\mathit{verify} :: \mathit{PublicK} \to \mathit{Natural} \to \mathit{Natural}\\&\mathit{verify}\ \mathit{pub}\ s = (s \uparrow (\mathit{pubE}\ \mathit{pub}))\ `\mathit{rem}`\ (\mathit{pubN}\ \mathit{pub})\end{aligned}$$

In this form, RSA is not safe, however. There are a lot of issues that must be addressed. In particular, we omitted everything related to *padding* messages before they are encrypted and to *hashing* messages before they are signed. These steps are essential for RSA to be secure. But since these steps have little relation to the mathematics of primes, they are not relevant to this chapter. After all, this is not an introduction to cryptography.

1.12 Open Problems

In the summer of 1742, Christian Goldbach (1690 – 1764), a mathematician and diplomat in the service of the Russian Czar, wrote a letter to Leonhard Euler. In this letter he told Euler of an observation he made: every even natural number greater than 2 can be represented as a sum of two primes. He had tested this with a lot of numbers – basically, with any number he could find – but was unable to prove the conjecture. Euler was excited about the observation and answered that he was sure that the conjecture must be true but that he was unable to demonstrate it either. This, basically, is still the state

of affairs today: There is a lot of evidence that Goldbach's conjecture is true, much more than in the times of Euler and Goldbach, but there is no rigorous proof.

Let us see some examples:

$4 = 2 + 2$,
$6 = 3 + 3$,
$8 = 3 + 5$,
$10 = 3 + 7 = 5 + 5$,
$12 = 5 + 7$,
$14 = 7 + 7 = 3 + 11$,
$16 = 3 + 13$

and so on.

Here is a function to compute for any number n one of the sums of two primes that equal n:

```
goldbach :: Natural → (Natural, Natural)
goldbach n | odd n      = error "not even"
           | n ≡ 2      = error "not greater 2"
           | otherwise = go allprimes
  where go [] = error "The end is ny!"
        go (p : ps) | p > n − 2 = error ("disproved: " ++ show n)
                    | otherwise = let q = n − p
                                  in if prime q then (p, q)
                                                else go ps
```

The really interesting point is the second error in the *go* function. When you can find a number, for which this error occurs, there is certainly some mathematical honour you can earn.

One can think of many ways to find numbers to which to apply the *goldbach* function, the result is always a prime tuple, *e.g.* powers of 2:

$(2, 2), (3, 5), (3, 13), (3, 29), (3, 61), (19, 109), (5, 251), (3, 509), (3, 1021)$,

the year of the first known formulation of the conjecture: $goldbach\ 1742 = (19, 1723)$, safe primes: $map\ goldbach\ (map\ (+1)\ safeprimes)$:

$(7, 257), (11, 337), (7, 353), (5, 379), (5, 463), (13, 467), (5, 499), (7, 557), (11, 577), \ldots$

or the ASCII code of the word "goldbach":
$goldbach\ \$\ read\ \$\ concatMap\ (show \circ ord)$ `"goldbach"`: ... (This will take a while to compute, so let us continue ...)

There is so much evidence in favour of the correctness of the conjecture that it is considered to be true by most mathematicians today. But there is still no rigorous proof. We cannot claim the truth of the conjecture just by applying it to a finite set of numbers,

even if that set is incredible large. The point is of course that there are infinitely many numbers. For any number n up to which we may have tested the property, there may be a larger number $n+k$ for which the property does not hold. By just computing concrete results for given numbers, we can therefore not prove the theorem; we can only disprove it this way by finding a number n for which the property does not hold.

The Goldbach property, *i.e.* the property of a number to be representable by the sum of two primes, is so prototypical for many problems in mathematics that properties of this kind are often called *Goldbach-like* in mathematical logic. Goldbach-like properties can be easily calculated for any given number n, but there is no way of proving that such a property holds for all numbers other than going through all of them and actually calculating it for each and every one. Since we cannot go through all numbers in a finite number of steps, this kind of statements, even though calculable for any given instance, cannot be calculated from the axioms of a given formal system.

There is nothing special about the Goldbach conjecture itself that would yield this characteristic. In fact, many unproven conjectures share it. That every number can be factored into primes, for instance, is a Goldbach-like statement too. For this statement, we actually have a simple proof. We even have a proof for the much stronger fundamental theorem of arithmetic that states that every number has a *unique* prime factorisation (which is not a Goldbach-like statement). If we did not have a proof of the fact that every number can be factored into primes, the only technique we would have at hand to prove or disprove the statement would be to find a counterexample. Testing numbers for this property as such would be quite simple, since we just have to factor given numbers and say whether they can be factored into primes or not. Until now, however, the property has turned out to be true for any number we have looked at. If we not had the proof of the theorem, we could imagine that there might be a counterexample lurking among the infintely many numbers we have not yet examined. But there are not enough computing resources to look at all of them in finite time.

A simple property that is not a Goldbach-like statement is the twin prime conjecture, which we already encountered, stating that there is an infinite number of pairs of primes p and q, such that $q = p + 2$. Examples are 3 and 5, 5 and 7, 11 and 13, 17 and 19, 29 and 31 and many other number pairs. Unlike the Goldbach conjecture, we cannot disprove the twin prime conjecture by finding a counterexample. The conjecture states that there are infinitely many twin primes. There is thus no criterion to conclude from the fact that a pair of numbers (n, m) does not have the property of being twin primes that the conjecture is false. We could, for instance, get excited about the fact that 23 has no twin and declare 17 and 19 the last pair of primes. However, when we go on, we will find 29 and 31. The only way to disprove the twin prime conjecture is therefore to go through all numbers, which, again, is not possible in a finite number of steps.

A similar is true for the fundamental theorem of arithmetic. We cannot decide this theorem even for a single number. For any number we factor, we additionally have to compare the obtained factorisation with those of all other numbers (including those we

have not yet examined) to decide if the factorisation is unique. Establishing the property for a single number already includes infinitely many steps and is therefore not possible in practical terms. Apparently, there is no way to prove this theorem but in an indirect fashion.

Another complex of open problems is factoring itself for which, as we have seen, no efficient algorithm is known. There are ways to find the prime factors of a given number, of course. But with large numbers, factoring becomes resource-intensive. We need a lot of steps to find the factors.

The factoring problem is tightly coupled with the problem of solving the discrete logarithm. We got an idea of this coupling already, when we looked at the Carmichael theorem: when we know the factors of a number, we can easily find a number t, such that $a^t \equiv 1 \pmod{n}$, where a and n are coprime. Shor's quantum factoring algorithm, actually, exploits this coupling the other way round. It finds the number t and uses t to find two prime factors of n.

In the world of classic, non-quantum computing, we know of only one way to find t, the order of the group generated by a, namely to raise a to the sequence of numbers $1, 2, \ldots, n-1$ until a raised to one of these numbers is 1 (mod n). Here is a Haskell function that uses this logic to find t for a given a:

```
order :: Natural → Natural → Natural
order n a = go 2 (a * a)
  where go t a' | a' `rem` n ≡ 1 = t
                | t ⩾ n          = error "group exhausted"
                | otherwise      = go (t + 1) (a' * a)
```

Note that we introduce a guard that saves us from looping eternally: when we reach $t = n$, we abandon the function. We will see why we do this in a second.

Once we have found t, it is easy to find the factors. We start – once again – with the equation

$$a^t \equiv 1 \pmod{n} \tag{1.70}$$

and subtract 1 from both sides yielding

$$a^t - 1 \equiv 0 \pmod{n}. \tag{1.71}$$

Then we factor the left-hand side of the equation:

$$(a^{\frac{t}{2}} - 1)(a^{\frac{t}{2}} + 1) \equiv 0 \pmod{n}. \tag{1.72}$$

In other words, the product of $a^{\frac{t}{2}} - 1$ and $a^{\frac{t}{2}} + 1$ is congruent to 0 modulo n, *i.e.* this

product is a multiple of n. That, in its turn, means that n and this product must have common factors. Consequently, $\gcd(a^{\frac{t}{2}} - 1, n)$ or $\gcd(a^{\frac{t}{2}} + 1, n)$ will produce at least one factor of n.

This does not work in all cases, though. First, t must be even; otherwise $\frac{t}{2}$ would not be a natural number. If t is odd, we therefore have to look for another a. Second, a must be coprime to n. On the other hand, If a is not coprime to n, then we have already found a factor of n, namely the gcd of n and a. Third, $a^{\frac{t}{2}}$ should be neither $n-1$ nor 1, since in that case 1.72 is trivally 0. We do not need to care about 1, though, because we already know that the size of the group is t, *i.e.* t is the first number such that $a^t \equiv 1 \pmod{n}$. But we have to check for $n - 1$.

Finally, n should be squarefree. If n is not squarefree, some numbers a will fulfil the equation $a^t \equiv 0 \pmod{n}$. That is, there are remainders of n that, multiplied by themselves, will yield a multiple of n. In that case, we will hit the error "group exhausted" in the *order* function above. The property of a number being squarefree or not, however, is hard to establish. If we introduce a test at the beginning of the algorithm, we are back where we started: an algorithm that takes an exponential number of steps. We therefore have to accept that an error may occur for any number on which we apply Shor's algorithm.

Here is a simple Haskell implementation of the classical part of Shor's algorithm using the order function defined above:

```
shorfact :: Natural → IO [Natural]
shorfact 0 = return []
shorfact 1 = return []
shorfact 2 = return [2]
shorfact n | even n     = do fs ← shorfact (n `div` 2)
                             return (2 : fs)
           | otherwise = do p ← rabinMiller 16 n
                            if p then return [n] else loop
  where loop = do a ← randomNatural (3, n − 2)
                  case gcd n a of
                       1 → check a (order n a)
                       f → do fs1 ← shorfact (n `div` f)
                              fs2 ← shorfact f
                              return (fs1 ++ fs2)
        check a t  | odd t = loop
                   | (a ↑ (t `div` 2)) `rem` n ≡ n − 1 = loop
                   | otherwise = let f1 = gcd n (a ↑ (t `div` 2) − 1)
                                     f2 = gcd n (a ↑ (t `div` 2) + 1)
                                     f | f1 ≡ 1     = f2
                                       | otherwise = f1
                                 in do fs2 ← shorfact (n `div` f)
                                       fs1 ← shorfact f
                                       return (fs1 ++ fs2)
```

As usual, we start with some trivial base cases: 0 and 1 have no factors; 2 is the first prime and has only one factor, namely 2 itself. Then, if n is even, we apply the algorithm to half of n and add 2 to the resulting list of factors. Otherwise, we first check if n is prime using the Rabin-Miller test (with a relaxed repetition value). If it is prime, there is only one factor, namely n. Otherwise, we enter *loop*.

Here, we start by choosing a random number that is not in one of the trivial groups of n and test if we have been lucky by checking if $\gcd(n, a) = 1$. If it is not 1, then we have found a factor by chance. We continue with *shorfact* on n dividied by f and on f, which still could be a composite factor of n. Finally, we merge the two resulting lists of factors.

Otherwise, we call *check*. This function decides how to continue depending on the result of *order*: we may need to start again, if t does not fulfil the preconditions, or we continue with the gcds. If t is odd, a turns out to be useless and we start again with another a. If $a^{\frac{t}{2}}$ is $n - 1$, a is again useless and we start with another a. Otherwise, we compute the gcds of $a^{\frac{t}{2}} \pm 1$. We take one of the results to continue just making sure that, should one of the values be 1, we take the other one. That is a bit sloppy of course, since the case that both results are 1 is not handled. If that happens, however, something must be wrong in our math and, then, we cannot guarantee that the result is correct at all.

The quantum magic enters in the *order* subroutine. The algorithm that finds the order of a uses techniques that are far from what we have learnt so far, in particular *Fourier analysis*. Fourier analysis is a technique that represents complicated functions in terms of simpler and well-understood functions, namely trigonometric functions. The idea is that the quantum processor is initialised with a *superposition* of the period of the function $f(x) = x^t \bmod n$, which is then simplified reducing f by Fourier analysis.

This sounds like alchemy – and, yes, basically it is alchemy, quantum alchemy to be specific. We will look at Fourier analysis later, when we have studied the calculus. But I cannot promise that you will understand quantum alchemy when you will have understood Fourier. I am not sure, in fact, if even quantum alchemists understand quantum alchemy.

The idea to apply Fourier analysis to finding prime factors leads to a fascinating view on numbers. Fourier analysis shows the simple wave functions that together lead to the complicated function we provide as input. In other words, each prime factor establishes a simple repetitive pattern; but since composite numbers have many prime factors, these patterns superpose each other, so they are hard to recognise. The subgroups of the group of remainders of a number are exactly these basic "waves" that together compose the main group and, as such, the composite number in question.

The result of the quantum Fourier analysis, as always in quantum computing, is correct with a certain probability. The whole algorithm, therefore, must be repeated and we must ensure that there is one result that appears significantly more often than others. The reasoning here is similar to the reasoning we have already applied to the Rabin-Miller test. So, this part of the algorithm (which we have left out above) should not be shocking. Shocking is rather the fact that there actually is a quantum algorithm that solves mathematical problems that cannot be solved (yet) on classic computers.

There may even be a hard boundary that cannot be passed, so that we have to accept that there are problems that can be solved in the classic world and others than can be solved only in the quantum world. This is an open and, indeed, very deep question in modern mathematics and computer science. But let us come back to primes, which, for my taste, are spooky enough.

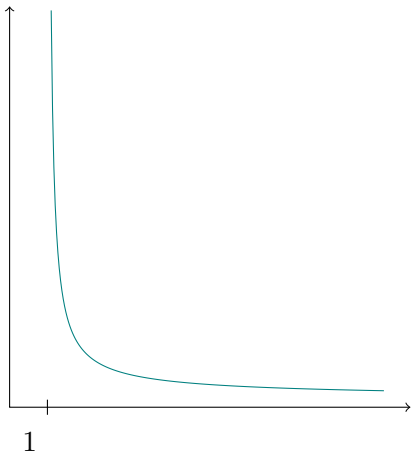
The most fundamental unsolved problems deal with the distribution of primes and the *prime number theorem*. To investigate the distribution of primes, we can start by counting primes up to a given number n, a function usually called $\Pi(n)$. We can implement Π in Haskell as:

$$\begin{aligned}&\mathit{countp} :: \mathit{Natural} \to \mathit{Int}\\&\mathit{countp}\ n = \mathit{length}\ (\mathit{takeWhile}\ (<n)\ \mathit{allprimes})\end{aligned}$$

$\Pi(10)$ or, in Haskell, *countp* 10 gives 4, since there are 4 primes less than 10: 2, 3, 5 and 7. Here are the values for some small powers of 10:

10^1	=	4
10^2	=	25
10^3	=	168
10^4	=	1229
10^5	=	9592

Such tables of values of Π were already compiled in the early 18^{th} century. From some of such tables Legendre conjectured that Π is somehow related to the natural logarithm (a beast we already met several times). Specifically, he proposed that $\Pi(n) = \frac{n}{\ln(n)+c}$, where c is some constant. Gauss and later his student Peter Gustav Lejeune Dirichlet (1805 – 1859) improved this conjecture with the *logarithmic integral*, a concept from calculus and, as such, far ahead on our way. Intuitively, the integral yields the area under a curve; the logarithmic integral is the area under the curve described by the function $\frac{1}{\ln(n)}$, which looks like this:



Let us compare the precision of these two approximations. The following table lists the values for $\Pi(n)$ and the differences $\frac{n}{\ln(n)} - \Pi(n)$ and $li(n) - \Pi(n)$, where $li(n)$ is the logarithmic integral of n:

n	$\Pi(n)$	$n/\ln(n)$	$li(n)$
10	4	0	2
10^2	25	-3	5
10^3	168	-23	10
10^4	1229	-143	17
10^5	9592	-906	38
10^6	78498	-6116	130
10^7	664579	-44158	339
10^8	5761455	-332774	754
10^9	50847534	-2592592	1701
...	...	...	...

The error of the logarithmic integral grows much slower than that of Legendre's conjecture. For small numbers, the error of the li variant is still greater. But already for $n = 10^3$, Legendre overtakes li and is then increasing orders of magnitude faster than li.

This is not the last word, however. There is a function that is still more precise in most cases. In his famous paper, "On the Number of Primes less than a given Magnitude", the ingenious mathematician Bernhard Riemann (1826 – 1866) not only introduced the most tantalizing of all unsolved mathematical problems, the *Riemann Hypothesis*, but he also proposed a refinement to Gauss/Dirichlet's Π-approximation. He conjectured that Π corresponds roughly to the infinite series

$$R(n) = li(n) - \frac{1}{2}li(n^{\frac{1}{2}}) - \frac{1}{3}li(n^{\frac{1}{3}}) - \frac{1}{5}li(n^{\frac{1}{5}}) + \frac{1}{6}li(n^{\frac{1}{6}}) - \dots \tag{1.73}$$

That the number of primes is related to the natural logarithm is already an astonishing fact. But, now, Riemann goes even further. To see what it is that is so surprising about Riemann's improvement, we present the equation above in a more compact form:

$$R(n) = \sum_{k=1}^{\infty} \frac{\mu(n)}{k} li(n^{\frac{1}{k}}) \tag{1.74}$$

The function $\mu(n)$ appearing in the formula is the Möbius function, which yields 0, 1 or -1 depending on n being squarefree and the number of prime factors of n being even or odd. To remind you of the first values of the Moebius function: $\mu(1) = 1$, $\mu(2) = -1$, $\mu(3) = -1$, $\mu(4) = 0$, $\mu(5) = -1$ and $\mu(6) = 1$ leading to the first terms of the summation above: $\frac{1}{1}li(n^{\frac{1}{1}}) = li(n)$, $\frac{-1}{2}li(n^{\frac{1}{2}})$, $\frac{-1}{3}li(n^{\frac{1}{3}})$, $\frac{0}{4}li(n^{\frac{1}{4}}) = 0$, $\frac{-1}{5}li(n^{\frac{1}{5}})$ and $\frac{1}{6}li(n^{\frac{1}{6}})$. In other words: the Möbius function is intimately related to the concept of the distribution of primes through the prime number theory, even if it does not reveal any regularity at the first and even the second sight.

For many values of n, the Riemann refinement is much closer to the real value of Π than either Legendre's try and Gauss/Dirichlet's improvement. It is esitmated that it

is better 99% of the time. Occasionally, however, Riemann's value is worse than that of Gauss/Dirichlet. The latter fact is known only theoretically, no specific n is known that makes Riemann worse. In most cases, Riemann's value is even much better, for instance: for 10^9, Gauss/Dirichlet's deviation is 1701 while Riemann's difference is only -79; The difference of Gauss/Dirichlet for 10^{16} is more than 3 million; Riemann's difference is 327 052 and thus 10 times better.

The prime number theorem is usually stated as an asymptotic law of the form:

$$\lim_{n\to\infty} \frac{\Pi(n)}{n/\ln(n)} = 1, \tag{1.75}$$

which means that for large n, the relation of $\Pi(n)$ and $\frac{n}{\ln(n)}$ approximates 1. A first attempt to prove the theorem was made by the Russian mathematician Pafnuty Lvovich Chebyshev (1821 – 1894). Even though his paper failed to strictly prove the theorem, he could prove Bertrand's postulate, which states that there is at least one prime number between n and $2n$ for $n \geq 2$ and on which we have already relied when looking at cryptography. A proof was finally given at the the end of the 19^{th} century independently by the French mathematicians Jacques Hadamard (1865 – 1963) and Charles Jean de la Vallée-Poussin (1866 – 1962).

It is unknown until today, though, if strict error margins for the approximation of Π can be given and if the available approximations can be further improved. Many of these questions are related to Riemann's Hypothesis. But mathematicians today appear to be far from producing a proof (or refutation) of this mega-hypothesis.