Lean 4 Quickstart

Lean 4 for mathematicians
Version 1, 01.24
Yves Jäckle

This is a tutorial to Learn how to use Lean 4 to verify mathematical content. It is intended for people learning Lean in Berlin. Please send me feedback about this tutorial! Did you find errors ? Was something incomprehensible or unclear ? What was your misconception and what caused it? Is there something you would like to add to the tutorial for future generations ? We can also meet in person, depending on my schedule.
Code for this resource may be found on: https://github.com/Happyves/BerLean-Quickstart/tree/main

Contents

1 Examples

1.1 Basics

The corresponding file begins with one of the most important formalization concepts: comments. Comments are used to make reading Lean code easier and we recommend you use comments generously throughout your projects.

Comments
If *text* were to represent your comment, then the syntax `/- text -/` will produce such a comment. With this syntax, you may jump lines in the comment. The syntax `- text` (2 minusses, though LaTeX only prints one) allows to make comments that fit only on the rest of the line.

We'll ignore the lines starting with `import` and `open` for now. Instead, let's look at our first proof in Lean!

First proof

```
example (n m: ℕ) (hn : Even n) (hm : Even m) : Even (n+m) :=
  by
  rw [even_iff]
  rw [even_iff] at hn hm
  rw [add_mod]
  rw [hn, hm]
  rfl
```

Let's study the first line. Here `example` declares that we're about to prove a theorem we won't give name (as theorems can be given names, as you'll see in a moment). Next, `(n m : ℕ)` declares two natural numbers named n and m, which are object of interests in our theorem. They are followed by declarations `(hn : Even n)` and `(hm : Even m)`, which declare the assumptions that n and m are even numbers, and names these assumptions `hn` and `hm` respectively. Finally, sandwiched between the `:` and `:=` is the statement of the conclusion of our theorem, `Even (n+m)`, which states that $n + m$ is even too.

We may now start the proof of the theorem, and we do so with the keyword `by`, on the second line of the code. Now, click after the `by` with your mouse. To the top right of VS Code, you should see the so-called ***infoview***, which will display the so-called ***tactic state***:

Tactic state

```
1 goal
nm: ℕ
hn: Even n
hm: Even m
⊢ Even (n + m)
```

Above the ⊢, the objects and assumptions of our theorem are displayed, and to its right, you can see the conclusion of the theorem. The tactic state is used to keep track of your current assumptions, and the current statements you wish to prove. Click after each line of code that follows the `by`: you will see in the infoview that changes to assumptions and conclusions are made.

In the four lines of code that follow, each line starts with `rw`, which stands for `rewrite`. Indeed, we will rewrite assumptions and conclusion to equivalent statements during our proof. the first rewrite, so you can see in the infoview, has the effect of replacing the conclusion `Even (n+m)` by `(n + m) % 2 = 0`. In Lean, the symbol `%` denotes the remainder in the Euclidean division. So what has happened is that the desired conclusion, which stated that $n + m$ is even, has been replaced with the desired conclusion that $n + m$ has remainder 0 in the Euclidean division by 2. To achieve this, we have used a theorem named `even_iff`, which appears under the square brackets in the `rw` line.

If you hover above `even_iff` in VS Code, you will see a box containing `even_iff {n : ℕ} : Even n ↔ n % 2 = 0` appear. Now, if you *ctrl + click* on `even_iff`, VS Code will open the folder in which this theorem is proven. As you can see in VS code's file explorer, the theorem is proven in a file of Mathlib.
Mathlib is a library (code written by other people) that contains many useful basic theorems such as `even_iff`. As you can see from the file, `even_iff` is a theorem, and it is proven using even more elementary theorems from mathlib. How, deep do foundations go ? We'll answer this throughout the tutorial, but for now, just note that we're using legit theorems to prove the theorem we set out to prove.

For now, we'll get back to our proof, looking at the second `rw`. It's the same as the previous line, but now we've added `at hn hm`. This will rewrite the definition of even numbers at assumptions `hn` and `hm`, as the syntax suggests. The third `rw` rewrites the conclusion again, but using a different theorem. Remember to hover above the theorems to see what they state: `add_mod (a b n : ℕ) : (a + b) % n = (a % n + b % n) % n`.
Here, Lean understands that it should take a to be n, b to be m and n to be 2. Then our goal has the shape of the left side of `add_mod`'s conclusion, and it replaces our conclusion with the right side.
We're almost done. First, notice that after the third rewrite, the goal is `(n % 2 + m % 2) % 2 = 0` and that our assumptions provide the values of `n % 2` and `m % 2`. Thus, with the fourth rewrite `rw [hn, hm]`, we use our assumptions to insert these values in the current goal.
At this stage, the goal is `(0 + 0) % 2 = 0`, and it amounts to a computation (recall that `%` takes the remainder), which is handled by the line `rfl`. Click after it to watch the infoview display "No goals", which means that our proof is complete !

Before moving on to another example, we present the following alternative proof, which illustrate how `rw` may be used. We may - or not - rewrite using multiple theorems or locations at a time.

rw alternatives

```
example (n m: ℕ) (hn : Even n) (hm : Even m) : Even (n+m) :=
  by
  rw [even_iff, Nat.add_mod]
  rw [even_iff] at hn
  rw [even_iff] at hm
  rw [hm]
  rw [hn]
  rfl
```

Next, consider the following formalized theorem and its proof. We've given it a name, `my_lemma`, as we'll use it in a moment, at which stage we'll refer to it with that name, just like we refer to mathlib theorems by their names. Can you say in words what we're proving ?

Second example

```
lemma my_lemma (n m: ℕ) (hn : Odd n) (hm : Odd m) : Odd (n*m) :=
  by
  rw [odd_iff] at *
  apply odd_mul_odd
  · exact hn
  · exact hm
```

The first line of the proof introduces one last way to use `rw`. The `*` symbol is used to denote "all possible options" in Lean, and in many other programming languages. Thus, the effect of the first line is to apply the rewrite at all assumptions and to the current goal, as the rewrite applies to all of them.

The line that follow is new to us. Hover above `odd_mul_odd` to find out that it states: `Nat.odd_mul_odd {n m : ℕ} (hn : n % 2 = 1) (hm : m % 2 = 1) : n * m % 2 = 1`. This translates to the statement "given natural numbers n and m, if both are odd, then their product is odd". Now, if you click in front of the `apply` in the code, you'll see in the infoview that the current goal matches the conclusion of the theorem `odd_mul_odd`. What the syntax `apply` does, is that it declares that we'll prove out goal by `apply`ing the theorem (`odd_mul_odd`, in this example). Thus, all that is left to prove are the assumptions to `odd_mul_odd`, as the proof will then be complete. This is why, as you can see be clicking after then end of the `apply` line, we end up with two new goals, one for each assumption of `odd_mul_odd` to be proven in our context. I personally find the name `apply` misleading: it helps me to think of it as "goal follows from ..."

After the `apply` line, we have two goals. To handle each separately, we use an indentation and the dot notation. Hover above the dot to read up on how to enter it with your keyboard. This has the effect that the infoview will focus on each goal individually, to make the infoview more readable. Now, each of the new goal actually corresponds exactly to the assumptions in our example. This is why we prove them with the `exact` syntax, displaying the corresponding assumptions.

There are shorter ways for performing this last part of telling Lean to use the assumptions to conclude:

Short syntax

```
example (n m: ℕ) (hn : Odd n) (hm : Odd m) : Odd (n*m) :=
  by
  rw [odd_iff] at *
  apply odd_mul_odd <;> assumption
```

Here, `<;>` asks Lean to try to prove all remaining goals with the code that follows, which in our case is `assumption`, a command ("tactic") that will tell Lean to prove the goal by using exactly the assumptions.

Now, let's see our lemma it action. We'll use it in the following example:

Using lemmata

```
example (n : ℕ) (hn : Odd n)  : Odd (n*3) :=
  by
  apply my_lemma
  · exact hn
  · rw [odd_iff]
    rfl
```

Again, the `apply` line instructs Lean that we'll show that the goal follows from our lemma `my_lemma`. Here, Lean recognizes that we want to use `my_lemma` with $m = 3$. The first sub-goal follows from the assumptions, while the second requires a bit more work. Take note of the indentation, which serves as example on how to prove sub-goals.

Next, we'll look at on more way to use lemmata, which I call "composite lemmata" (not an official name):

Composite lemmata

```
example (a : ℕ) (ha : Odd a)  : Odd (a*3) :=
  by
  apply my_lemma a _ ha
  rw [odd_iff]
  rfl
```

The composite `my_lemma a _ ha` will be interpreted by Lean as a theorem stating `(h : Odd 3) : Odd (a*3)`, which is why the `apply` will produce only one goal, corresponding to the single hypothesis of the composite theorem. What is happening in `my_lemma a _ ha` is that what follows `my_lemma` is interpreted as the inputs (objects and assumptions) to `my_lemma`, in the order in which they are listed in the statement of `my_lemma`. First, `my_lemma a` tells Lean to use `my_lemma` with $n = a$. Next, instead of the underscore, we could have written 3. Hover, I want to illustrate that Lean can infer some of the inputs from context. It will see that `my_lemma`'s m should be 3, by matching the goal from the current proof to the conclusion of `my_lemma`. Play around with the underscore (also called *wildcard*) to see what can be inferred by Lean. Finally, displaying `ha` in the input to `my_lemma` avoids us from getting it as a goal after the `apply`. So, to conclude `my_lemma a 3 ha` is a lemma which will have the conclusion of `my_lemma`, for values $n = a$ and $m = 3$, and the only assumption that 3 is odd, since the assumption that a is odd has already been satisfied through `ha`. To get a good sense for what is going on, click on the `check` of each of the following lines and refer to the infoview, which will display the meaning of each of these composite lemmata:

Checks

```
#check my_lemma
#check my_lemma 3
#check my_lemma 3 5
#check my_lemma 3 5 proof_of_oddity_3
#check my_lemma 3 5 proof_of_oddity_3 proof_of_oddity_5
```

Now, we will introduce `have` and illustrate composite lemmata and `exact`.

have

```
example (n : ℕ) (hn : Odd n)  : Odd (n*3) :=
  by
  have H : Odd 3 :=
    by
    rw [odd_iff]
    rfl
  exact my_lemma n 3 hn H
```

Here, `have H : Odd 3 :=` will declare a temporary lemma that will only be accessible within the proof of the example. `H` is its name, `Odd 3` is its conclusion, and it inherits the objects and assumptions from the proof the `have` is in, though we make no use of this in this example. We prove this temporary lemma with the same syntax as for normal proofs, except that we've indented lines. Clicking after `rfl` will display "No goals" which means that the `have`, not the actual statement we're working on, has been proven. Then, clicking in front of the `exact` will let you see that we've gained a new fact in our proof state, named `H` and stating `Odd 3`.
Since we have all inputs to `my_lemma` available, we may get its conclusion as a composite lemma `my_lemma n 3 hn H`, which matches the goal of our main proof, so that we may conclude it with an `exact`. The following variation allows us to follow this last part more clearly, and also displays the fact that we may omit the conclusion of a `have` in certain cases where Lean can infer it.

have without conclusion

```
example (n : ℕ) (hn : Odd n)  : Odd (n*3) :=
  by
  have H : Odd 3 :=
    by
    rw [odd_iff]
    rfl
  have Goal := my_lemma n 3 hn H
  exact Goal
```

There are many more commands ("tactics") to be learned, but `rw, apply, exact, have` cover the most important ones. We will now pick up the pace a bit. Remember to consult the infoview and to hover above theorems and commands to get a more detailed description of what they do.

Next up is working with universal quantifiers and implications:

∀ and →

```
example : ∀ n : ℕ, Odd n → Odd (n*3) :=
  by
  intro n hn
  exact my_lemma n 3 hn proof_of_oddity_3

example (n : ℕ) (hn : Odd n)  : Odd (n*3) :=
  by
  revert hn
  intro H
  exact my_lemma n 3 H proof_of_oddity_3

example (a : ℕ) (ha : Odd a) (fact : ∀ n m : Nat, Odd n → Odd m → Odd (n*m)) : Odd (a*3) :=
  by
  specialize fact a
  specialize fact 3 ha
  exact fact proof_of_oddity_3
```

The first example illustrates how to *intro*duce objects and assumptions to the proof state. In the second example, we display the reverse command to `intro`, which is `reverse`. As the last example illustrates, when a universally quantified proposition or an implication are in the form of assumptions, we can *specialize* them to specific values and cases, similarly to how we worked with composite lemmata.

∃ and an aspect to `rw`

```
example (n : ℕ) : ∃ m : ℕ, n < m :=
  by
  use n+1
  apply lt_succ_self

example (n : ℕ) (h : ∃ m, n = m+m) : Even n :=
  by
  cases' h with m eq
  rw [← two_mul] at eq
  rw [even_iff_exists_two_mul]
  use m

example (n : ℕ) (h : ∃ m, n = m+m) : Even n :=
  by
  obtain ⟨m , eq⟩ := h
  rw [← two_mul] at eq
  rw [even_iff_exists_two_mul]
  use m
```

In the first example, we show existence of an object by displaying a concrete such object using the `use` keyword. Then, the claimed properties of the object remain to be proven. Next, in the second example, if we assume existence of an object, then we may name the object and its property with the `cases'` *assumption* `with` *object-name property-name* syntax.

The ← in an `rw` before a theorem stating an equivalence or an equality will have the command reverse the order in which they consider the equivalence or equality (the default is left to right). So in the second example, the `rw` will consider $n + n = 2n$ instead of $2n = n + n$. The third example is an alternative to the second one, where syntax `obtain` ⟨ *object-name*, *property-name* ⟩ `:=` *assumption* does the same as `cases'`.

Conjunctions, the fancy word for "and", are handled as follows:

∧

```
example (n : ℕ) : ∃ m : ℕ, (n ≤ m ∧ Even m) :=
  by
  use 2*n
  constructor
  · apply le_mul_of_pos_left
    decide
  · rw [even_iff_exists_two_mul]
    use n

example (a b c : ℕ) (H : a ≤ b ∧ b ≤ c) : a ≤ c :=
  by
  apply @le_trans _ _ a b c
  · exact H.left
  · exact H.right
```

To prove a conjunction, as we're brought to do in the first example, must show each of the two properties. The `constructor` command allows us to obtain separate goals for each. Note the use of `decide` to prove a computation. When a conjunction is in the assumptions, we may access each of its sides by appending `.left` (or `.1`) or `.right` (or `.2`) to its name. Ignore the in the apply line for now.

Disjunctions, the fancy word for "or", are handled as follows:

∨

```
example (n : ℕ) (h : ∃ m, n = 2*m) : n = 42 ∨ Even n :=
  by
  right
  rw [even_iff_exists_two_mul]
  exact h

example (n : ℕ) (h : n = 42 ∨ Even n) :  Even n :=
  by
  cases' h with h42 he
  · rw [h42]
    decide
  · exact he
```

To prove a disjunction, we only need to prove one of its sides. The side we want to show is selected with commands `left` or `right`. If we assume a disjunction, as in the second example, we must produce a proof for each of its cases. Here too we may use the `cases'` command and syntax to perform this task. Note the use of `decide`: "Even" is actually defined to be the property of having remainder 0 in the division by 2, so checking "Even 42" amounts to computation.

Finally, we shall discuss equivalences:

```
↔
```

```
example (n : ℕ) : (∃ m, n = m+m) ↔ Even n :=
  by
  constructor
  · intro h
    cases' h with m eq
    rw [← two_mul] at eq
    rw [even_iff_exists_two_mul]
    use m
  · intro h
    rw [even_iff_exists_two_mul] at h
      obtain ⟨m , eq⟩ := h
    use m
    rw [← two_mul]
    exact eq

example (n : ℕ) : (∃ m, n = m+m) ↔  Even n :=
  by
  rw [even_iff_exists_two_mul]
  constructor <;> { intro h ; convert h using 2 ; rw [two_mul] }

example (n : ℕ) : (∃ m, n = m+m) ↔  Even n :=
  by
  congr! 3

#check (even_iff_exists_two_mul 2)
#check (even_iff_exists_two_mul 2).mp
#check (even_iff_exists_two_mul 2).mpr
```

Proving an equivalence amounts to proving an implication and its reverse. Seeing as $A \leftrightarrow B$ is $(A \to B) \wedge (B \to A)$ in disguise, we may use `constructors` to proceed. As you can see from the second and third examples, a good knowledge of available commands can shorten proofs drastically (though often at the expense of readability).
Finally, the last few `checks` illustrate how to use `.mp` (for *m*odus *p*onens) and `.mpr` (for *m*odus *p*onens *r*everse) to turn equivalences into implications.

There are many more commands to be discussed. In fact, one can make custom commands, so listing them would practically never be exhaustive. We just illustrated some of the more basic and important ones, and we plan on making a list of useful such "tactics" in the "Tactics" chapter. In what follows, remember that hovering above theorems and commands displays a more detailed description of what they do.

1.2 Infinitude of primes

Now, we'll look at a more sophisticated proof. Our goal is to formalize:

Infinitude of primes
There are infinitely many prime numbers.

Proof: Assume, for contradiction, that there exist only finitely many prime numbers $p_1, ..., p_n$.
We can consider their lcm $\prod_{i=1}^{n} p_i$, and its successor $\left(\prod_{i=1}^{n} p_i\right) + 1$.

On the one hand, any number has a prime divisor, so that there is some prime p_i that divides $\left(\prod_{i=1}^{n} p_i\right) + 1$.

This prime also divides the product $\prod_{i=1}^{n} p_i$, as it's one of its factors.

Yet, since a number and its successor are coprime (hence, have as only common divisor 1), the prime p_i would equal 1, which isn't prime ! We've reached the desired contradiction. □

To formalize this, we'll elaborate it a bit. We'll first show that for any finite set s, there exists a prime p not contained in s. Then, we'll derive the mathlib notion of infinity from this latter statement. So we first look at:

First part

```
theorem Euclid_proof : ∀ (s : Finset ℕ), ∃ p, Nat.Prime p ∧ p ∉ s :=
  by
  intro s
  by_contra! h
  set s_primes := (s.filter Nat.Prime) with s_primes_def
  -- Let's add a membership definition lemma to ease exposition
  have mem_s_primes : ∀ {n : ℕ}, n ∈ s_primes ↔  n.Prime :=
    by
    intro n
    rw [s_primes_def, mem_filter, and_iff_right_iff_imp]
    exact h n
  -- In order to get a prime factor from 'nat.exists_prime_and_dvd', we need:
  have condition : (∏ i in s_primes, i) + 1 ≠ 1 :=
    by
    intro con
    rw [add_left_eq_self] at con
    have however : 0 < (∏ i in s_primes, i) :=
      by
      apply prod_pos
      intro n ns_primes
      apply Prime.pos
      exact (mem_s_primes.mp ns_primes)
    apply lt_irrefl 0
    nth_rewrite 2 [← con]
    exact however
  obtain ⟨p, pp, pdvd⟩ := (exists_prime_and_dvd condition)
  -- The factor also divides the product:
  have : p ∣ (∏ i in s_primes, i) :=
    by
    apply dvd_prod_of_mem
    rw [mem_s_primes]
```

```
    apply pp
  -- Using the properties of divisibility, we reach a contradiction thorugh:
  have problem : p ∣ 1 :=
    by
    convert dvd_sub' pdvd this
    rw [add_tsub_cancel_left]
  exact (Nat.Prime.not_dvd_one pp) problem
```

First, we apply rule nr. 1 of coding: keep calm, this is fine.
The proof is long, and there is a lot of new syntax in it. We'll discuss it step by step, in order of appearance. Remember to follow the proof in the infoview, and to hover above theorems and commands to see what they are and do. Here we go:

- `(s : Finset ℕ)` means that s is a finite set of natural numbers
- `intro` is used to *intro*duce objects or assumptions from the current goal statement to the current assumptions of the proof state. It works with universal quantifiers $\forall$, implications $\to$ and also negative statements, where it amount to a proof by contradiction. For the latter use, consider the "`condition`" `have`-lemma of the above proof as example. What follows intro will be the names of the objects/assumptions introduced to the proof state.
- `by_contra!` allows for proof by contradiction, where the assumption of the negation of the goal will be named with what follows the command.
- The syntax `set` *name1* `:=` *object* `with` *name2* will produce an object named *name1* with value *object*, and will introduce fact `name2 :` *`name1 = object`* to the proof state.
- `s.filter` will produce a finite set that is obtained by taking the elements of s that satisfy the property that follows. In the proof above, it's the property of being prime.
- The notation in the "`condition`" `have`-lemma is how one writes products over finite sets.
- Writing `.mp` after a theorem that states an equivalence $\leftrightarrow$ will produce an implication $\to$ version of this theorem.
- `nth_rewrite` is like `rw`, except that its first input is the location of the pattern we wish to rewrite in the target pattern. In the above proofs, there are two 0 we would like to replace with the product, via assumption `con`, and `nth_rewrite` allows us to target the correct one.
- `convert` is like `apply` except that the goal of the theorem that follows doesn't have to match the current goal of the proof. Instead, Lean will a more relaxed matching of the expressions. In the above proof, a simple `apply` would fail, as Lean fails to match the 1 from the current goal with the subtraction from `apply`ed lemma. `convert` on the other hand will set this matching as a new goal in the form of an equality.

Now, we will derive the infinitude of the set of primes from our previous result:

Second part

```
lemma Euclid_proof_standardised : {n : ℕ | Nat.Prime n }.Infinite :=
  by
  rw [Set.Infinite]
  intro con
  obtain ⟨p, ⟨p_prop, p_mem⟩⟩ := Euclid_proof (Set.Finite.toFinset con)
  apply p_mem
  rw [Set.Finite.mem_toFinset con]
  dsimp
  exact p_prop
```

We take note of the notation `{ x :  X | P(x) }` to define sets, and of the use of `dsimp` (for *d*efinitional *simp*lification) to rephrase $y \in \{x : X | P(x)\}$ to $P(y)$, in the above proof. Generally speaking, `dsimp` will simplify statements to their most basic form.

Now, we will ask you to try out some Lean theorem proving. We will again prove the infinitude of primes, following a different proof this time, based on the properties of so-called Fermat-primes. We first recall the informal proof.

Proof: We consider the Fermat numbers $F_n = 2^{(2^n)} + 1$, and show that they satisfy the recursive relation $\prod_{i=0}^{n} F_i = F_{n+1} - 2$ with base term $F_0 = 3$, by induction.
For the base case, we note that indeed, $F_n = 2^{(2^0)} + 1 = 2^1 + 1 = 3$.
For the step, $\prod_{i=0}^{n} F_i = F_n \prod_{i=0}^{n-1} F_i = F_n(F_n - 2) = \left(2^{(2^n)} + 1\right)\left(2^{(2^n)} - 1\right) = 2^{(2^{n+1})} - 1 = F_{n+1} - 2$.
The relation $\prod_{i=0}^{n} F_i = F_{n+1} - 2$ implies that distinct Fermat numbers are coprime.
Indeed, a common divisor to F_i and F_j, where wlog $i < j$, divides $\prod_{k=0}^{j-1} F_k = F_j - 2$ as it divides F_i, which is in product $\prod_{k=0}^{j-1} F_k$: it therefore divides 2. Yet, since Fermat numbers are odd, the divisor can't be 2, so it can only be 1.
The Fermat number therefore are a sequence of pairwise coprime numbers. If for each such number, we consider a prime divisor, then we get a sequence of prime numbers. To conclude that there are infinitely many primes, we must show that the primes from the latter sequence are pairwise different.
If they weren't, they'd be a common divisor to two distinct Fermat numbers, which are coprime, so that this prime divisor would be 1, which the desired contradiction.
This allows us to conclude that there is an infinite sequence of distinct prime numbers. □

We'll guide you through the formalization, letting you fill out certain parts.
First, we define the Fermat numbers by defining a *fun*ction that given n returns the nth Fermat number.

Fermat numbers

```
def F : ℕ → ℕ := (fun n => 2^(2^n) + 1)
```

We take note of the syntax for defining functions.

In our file, this definition is preceded by a so-called *doc-string*.

Doc-string
A *doc-string* is a comment explaining the object/theorem that is follows.
Use syntax `/-` (two minuses, though LaTeXdoesn't want to) *content* `-/` above the object/theorem to make a *doc-string*

By hovering over `F`, in the rest of the code, you will see the text we gave as doc-string appear.

Lean can actually compute Fermat number! Check the infoview on line `#eval F 7` to compute the 7th Fermat number. Just below that line is the first lemma we ask you to prove! As a hint, note that this lemma is purely computational. Delete the `sorry` which serves as a placeholder for proofs and causes the theorem to be a admitted. Keep in mind that this exercise and those that follow aim at teaching you when to use the right command. If mathlib lemmata are needed, we'll provide them for you.

Your next exercise will be to find the rest of the proof to lemma `fermat_stricly_monotone`. It says that Fermat numbers are increasing. Read the doc-string to find its purpose in the proof of the infinitude of primes as a whole. Below the lemma, we listed the theorems you'll need to perform the proof, in the order that they show up in the proof. Hover above them to see what they do. If you see `succ` appear in what follows, think of it as "+1" (it stands for *succ*essor and will be explain in the next chapter). Take note of the syntax for `dsimp` at the start of the proof.

Next, we have the same exercise format for `fermat_bound`. Take note of the `induction'` syntax: n is the number we perform induction on and it is also the name of the variable we'll use for the induction step ; `ih` is the name we'll give the induction hypothesis, when proving the step. From now on, you may have to use theorems we defined it this very file !

For the proof of `fermat_odd`, we've given you the theorem to use, but not in the order they appear in in our solution, to spice things up.

We'll let you get away with simply reading `fermat_product`. Important thing to note are : the `ring` tactic, which can prove certain algebraic identities on its own ; the syntax `.symm` to flip an equality, so that it matches the goal, for example ; the fact that certain operations with subtraction on naturals require care in Lean. To illustrate the latter, consider the result of `#eval 4 - 6`. We'll get to this in the next chapter.

In `fermat_coprimes`, there are some sorry's to fill out, using the theorems we have in the `check`s below. This time however, two theorems we used in our solution are missing among the `check`s: luckily, they already appeared in the same file! Take note of how one denotes coprime numbers, and the gcd (greatest common divisor). In the very last step, we use the `exfalso` principle, which states that any result follow from contradiction. We'll derive `False` from our assumption, so that we'll show that they're contradictory.

In `fermat_neq` we use the `linarith` tactic, which can solve certain problems of linear arithmetic for us.

Finally, your last exercise will be to read through our two concluding results, `second_proof` and `second_proof_standardised`, and understand what is going on. This may sound like a cheap exercise (it is), but it is also quite relevant. The solutions we provide merely add comments and doc-strings. When reading other people's code (including mathlib code), comments and doc-string may be missing (in fact, the `Classical.choose` used in the second proof has no doc-strings for example), and you'll have interpret what is going on purely from the code.

2 Naturals and integers

2.1 Basics on natural numbers

Natural numbers are defined as an inductive type in Lean:

Definition of the natural numbers

```
inductive nat where
| zero : nat
| succ (n : nat) : nat
```

This definition declares two objects: `nat.zero`, which will represent 0, and `nat.succ`, which is a function that represents the *succ*essor of a given natural number. The number 2 for example is encoded as `nat.succ (nat.succ nat.zero)`. It may seem weird that we called `nat.succ` a function, and that we provide it with an input n in its definition that isn't used to determine an output value. This is why the word "*constructor*" is prefered for inductive types. `nat.succ` isn't suposed to be computed, its job is to help encode natural numbers.

To see how this definition works in action, we'll next define addition of natural numbers:

Definition of addition of natural numbers

```
def nat.add : nat → nat → nat
| a, nat.zero   => a
| a, nat.succ b => nat.succ (nat.add a b)
```

It's a function that takes a left term (referred to with `a`) and a right term and does the following. It looks at the second term and disjoins cases on whether the it's 0 or the successor of a number (referred to with `b`). These are the only possibilities for a natural number, as we saw that they are constructed using these two *constructors*. In the case where the right term is 0, addition will return the value of the left term `a`, as expected. In the case where the right term is the successor of a number `b`, we use the conceptual $a + (b + 1) = (a + b) + 1$ to define the returned value to be the successor of the addition of `a` and `b`. To find the latter, we recursively call the addition function.

In out file, the infoview for line `#reduce nat.add two two` will return the result of $2 + 2$. Here, `reduce` will tell Lean to replace function applications by their value, until this isn't possible anymore. So the way Lean "computes" $2 + 2$ is by recognizing that it has form $2 + (1 + 1)$ (or $2 + (succ\ 1)$ by using hybrid notation), so that the second case of addition's definition applies, and we get to $(2 + 1) + 1$. The parenthesized part can still be reduced, as it's read as $2 + (0 + 1)$, so that the expression is *reduced* further to $((2 + 0) + 1) + 1$. Finally, the innermost part corresponds of addition's first case, and we obtain $(2 + 1) + 1$. Since 2 was interpreted as $(1) + 1$ in the first place, we've arrived at expression $(((1) + 1) + 1) + 1$, which represents 4.

Now, let's address the use of recursion. Lean allows us to define recursive functions:

Definition of the factorial

```
def factorial : ℕ → ℕ
| Nat.zero => 1
| Nat.succ n => (Nat.succ n) * (factorial n)
```

This is for example how one can define the factorial on a natural number (Lean's actual natural numbers, not the replica we've built from scratch so far). This recursion will end, as the term we recursively call the function on has "one less constructor" then the initial term the function was called on, and since terms are made up of finite applications of constructors, the calls will eventually stop. For more complicated recursive definitions, Lean will ask us to justify termination.

Now, how do we prove propositions about our natural numbers and their operation, such as $a + b = b + a$?
We shall first prove a few lemmata to be used in the proof of $a + b = b + a$.

1st lemma

```
lemma my_lemma_1 : ∀ (n m : nat), nat.add (nat.succ n) m = nat.succ (nat.add n m) :=
  by
  intro n m
  induction' m with m ih
  · dsimp [nat.add]
  · dsimp [nat.add]
    rw [ih]
```

This lemma states $(n + 1) + m = (n + m) + 1$. It is proved by induction on m. Here, induction is a principle applicable to inductive types, such as `nat`, in general! It says that if we have the properties for the type's constants, and the property is maintained when applying the type's constructors, then the property is true for all terms of the type. In our case, the base case (for the constant `nat.zero`), states $(n + 1) + 0 = (n + 0) + 1$. In both these additions, we may reduce addition according to it's first case, as the second argument is 0. This is achieved with the `dsimp`, which yields $n + 1 = n + 1$, so that there's nothing left to prove. For the induction step (corresponding to the unique constructor `nat.succ`), we want to show $(n + 1) + (m + 1) = (n + (m + 1)) + 1$, assuming that $(n + 1) + m = (n + m) + 1$. Again, the additions may be reduces, both according to addition's second case, to yield $((n + 1) + m) + 1 = ((n + m) + 1)) + 1$, using `dsimp`. Now, on both sides of the latter equality, what's inside the $(...) + 1$ is actually equal according to the induction hypothesis. So we may rewrite the induction hypothesis with `rw`, to obtain an equation with identical sides, so that Lean is satisfied and the proof is complete.

We ask you to prove next lemma we'll need, `my_lemma_2`, as an exercise.

We may now assemble the lemmata to prove our initial goal:

Commutativity of addition

```
example : ∀ (n m : nat), nat.add n m = nat.add m n :=
  by
  intro n m
  induction' m with m ih
  · dsimp [nat.add]
    rw [my_lemma_2 n]
  · dsimp [nat.add]
    rw [my_lemma_1]
    rw [ih]
```

Follow what is happening at each line in the infoview.

Next, how would we define relations on our natural numbers, such as $a \leqslant b$ for example ?
We may achieve this by defining inductive propositions:

$\leqslant$

```
inductive nat.le (n : nat) : nat → Prop
| refl     : nat.le n n
| step {m} : nat.le n m → nat.le n (nat.succ m)
```

In Lean, proofs of propositions are built up in a similar way to how we encoded natural number. We'll elaborate this in a later chapter. What the previous definition says, is that proofs of $a \leqslant b$ are built from constant ($\approx$ axiom) $n \leqslant n$ and constructors $a \leqslant b \Rightarrow a \leqslant b + 1$. In our file, we illustrate this by proving $0 \leqslant 2$ in our context.
Since $0 \leqslant 2$ is for form $0 \leqslant (1 + 1)$, it will follow from "axiom" `nat.le.step`, if we prove $0 \leqslant 1$ (as we can then set $a = 0$ and $b = 1$ in `nat.le.step`). Since the latter is of form $0 \leqslant (0 + 1)$, we may use the same appraoch to reduce the proof to that of $0 \leqslant 0$. This in turn, is given by `nat.le.refl`. In our file, you'll find this proof in both a step-by-step form, and in a form using composite lemmata.
We then ask you to show that for any natural numbers n and m, we have $n \leqslant n + m$ as an exercise.

To conclude our little tour of the axiomatics of natural numbers, we discuss subtraction. We do so by defining the predecessor of a natural number first:

Predecessors and subtraction of natural numbers

```
def nat.pred : nat → nat
| nat.zero   => nat.zero
| nat.succ a => a

def nat.sub : nat → nat → nat
| a, nat.zero      => a
| a, nat.succ b => nat.pred (nat.sub a b)
```

It has the effect that the subtraction $a - b$ will equal 0 if $a \leqslant b$ (try to prove this ?), and we give some examples of this in our file. This makes sense (in a sense), as integers haven't yet entered the picture. It highlights the fact that in Lean, contrary to informal maths, the difference between natural numbers and integers is a big deal! The next example illustrates that such small formal conventions can have an impact on theorems where we might not have expected an impact. We show that addition and subtraction don't associate, though we may taken this for granted in informal math, as we make no distinction between natural numbers and integers.

Associativity

```
example : ∃ a b c : ℕ, (a + b) - c ≠ a + (b - c) :=
  by
  use 1
  use 2
  use 3
  decide
```

Indeed, $(1+2)-3 = 0$ while $1+(2-3) = 1$, since $2-3$. As you can tell from hovering above the `Nat.add_sub_assoc` we included in te file, we have associativity, but only under additional assumptions.

2.2 Euclidean division

Before moving on to the integers, we'll investigate the Euclidean division in Lean.

Euclidean division
For any numbers n and m, there are numbers q and r with $r < m$ such that $n = qm + r$.

q is the *quotient* and r the *remainder*. The quotient is computed by repeatedly subtracting m from n until we can't, at which stage the number of iterations is the quotient, and what remains from the subtractions is the remainder:

$$n \underbrace{-m - m - ... - m}_{q \ times} = r$$

We may define the quotient via recursion in Lean:

Quotient

```
def quotient (n m : ℕ) : ℕ :=
   if h : 0 < m ∧ m ≤ n
   then
      have fact : n - m < n := Nat.div_rec_lemma h
      (quotient (n - m) m) + 1
   else
      0
```

The recursion occurs at line `(quotient (n - m) m) + 1`: the number of subtractions for n is one more then that of $n - m$. So what is all the other stuff for ? We need our recursion to terminate, so the recursive call must be to a smaller argument, so that we want $n - m < n$. This is ensured by $0 < m$, as well as $m \leqslant n$, as for $n = 0$, Lean's subtraction on naturals would invalidate $n - m < n$. Condition $m \leqslant n$ also allows us to handle the case where the dividend is smaller then the divisor, where we expect the quotient to be 0. So unless these two conditions aren't satisfied, we recursively call `quotient`, and otherwise we return 0. The `have` line is there to convince Lean that the recursion will terminate.There is an interesting side-effect of this definition: we may divide by 0, in which case the quotient is defined to be 0.
Next, we define the remainder in a similar computational way. Here, we recursively call the remainder until the dividend is smaller then the divisor, at which stage we reached the remainder:

Remainder

```
def remainder (n m : ℕ) : ℕ :=
   if h : 0 < m ∧ m ≤ n
   then
      have fact : n - m < n := Nat.div_rec_lemma h
      remainder (n - m) m
   else
      n
```

This has the side effect that the remainder is precisely the dividend, if we're dividing by 0.

Now we would like to formalize the theoretical statement of Euclidean division, in the sense that for quotient $q(n, m)$ and remainder $r()n, m)$, we have $n = q(n, m) * m + r(n, m)$. We'll first give a semi-formal proof.

Proof: First, we'll handle the side effects. If $m = 0$ then $q(n, m) = 0$ and $r(n, m) = n$, so the theorem holds in this unconventional case. For the case that $0 < m$, we'll show the result by strong induction on n. In the base case of $n = 0$, $q(0, m) = 0$ and $r(0, m) = 0$, and the theorem holds. Next, if the theorem is true for any $a < n$, we'll show that it holds for n. First, if we had $n < m$, we'd end up with the same case as for $m = 0$, so we move to the case where $m \leqslant n$. So we have the condition to make the recursive call, hence $q(n, m) = q(n - m, m) + 1$ and $r(n, m) = r(n - m, m)$. Since we have $n - m = q(n - m, m) * m + r(n - m, m)$as inductive assumption, by taking $a = n - m$, we may rewrite it to $n = (q(n - m, m) + 1) * m + r(n - m, m)$ and finally to the desired $n = q(n, m) * m + r(n, m)$ using the recursion relations. □

As will become usual for you, the formal version of the proof is a bit longer:

Euclidean division

```
example (n k : ℕ) : (remainder n k) + k * (quotient n k) = n :=
  by
  by_cases Q : k = 0
  · rw [Q]
    rw [zero_mul, add_zero]
    rw [remainder]
    have Ifs_false : ¬ (0 < 0 ∧ 0 ≤ n) :=
     by
     push_neg
     intro problem
     exfalso
     exact (Nat.lt_irrefl 0) problem
    rw [dif_neg Ifs_false]
  · apply (@Nat.strong_induction_on (fun x => remainder x k + k * quotient x k = x) n)
    intro p ih
    replace Q := Nat.pos_of_ne_zero Q
    by_cases K : k ≤ p
     · rw [remainder, quotient]
       rw [dif_pos ⟨Q,K⟩]
       dsimp
       rw [if_pos ⟨Q,K⟩]
       have le_lem : p - k < p := Nat.sub_lt_of_pos_le Q K
       specialize ih (p-k) le_lem
       rw [mul_add, mul_one, ← add_assoc]
       rw [eq_comm] at *
       exact Nat.eq_add_of_sub_eq K ih
     · rw [remainder, quotient]
       have Ifs_false : ¬ (0 < k ∧ k ≤ p) :=
        by
        exact not_and_of_not_right (0<k) K
       rw [dif_neg Ifs_false, dif_neg Ifs_false]
       rw [mul_zero, add_zero]
```

As for the infinitude od primes proof, we step through some novelties, by order of appearance.

- `by_cases` is used to start a proof by case disjunction. It's followed by the name of the property to be disjoined on, and its statement.
- In the first `have`, we use the symbol ¬ to represent negation.
- `push_neg` is a command that will turn the statement into a logically equivalent one, in which the negations cannot be "simplified" further. You may want to read its doc-string for the coming exercise.

- The statements `dif_pos`, `dif_neg` and `if_pos` and `if_neg` are used to replace `if then else` statements, when we know if their condition is satisfied or not. The "`d`-version", where "d" is for *d*ependent is due to type considerations. Ignore it for now. To tell which to use, hover above the `if` in the proof state in the infoview: if `dite` is displayed, use the "`d`-version".
- the version of strong induction we use states `Nat.strong_induction_on {p : ℕ → Prop} (n : ℕ) (h : ∀ (n : ℕ), (∀ m < n, p m) → p n) : p n`. The reason we use in the line where we `apply` it is because, Lean can't tell what the input `p` is supposed to be, thought it's supposed to infer it (= guess it from context), as is signaled by the use of curly braces for this input.
- `replace` does exactly what its name suggests.

You can think of how to shorten the latter formal proof. The mathlib version of Euclidean division is `Nat.mod_add_div`.

Now it's your turn to work with remainders. We ask you to show that the remainder is less ten the divisor by filling out the `sorry` in our file. All the tools you'll need were presented in the previous proof. Though we must warn you: there is a little (?) mistake in our statement ! In fact, the mistake can even be found in the theorem: did you spot it before arriving to this paragraph ?

2.3 Basics on integers

Now, we'll quickly discuss integers and their relation to natural numbers. Lean's integers are defined as:

Integers

```
inductive int where
| ofNat   : ℕ → int
| negSucc : ℕ → int
```

We get non-negative integers, which represent the natural numbers, via `int.ofNat`, and negative integers via `int.negSucc`, which will be *thought of as* $n \mapsto -n-1$. A construction of integers as an inductive type defined by a constant 0 and two constructors, one for a successor and one for a predecessor is doomed, as it wouldn't allow for a unique representation of integers. We would have, as a Gedankenspiel, both `0` and `succ (pred 0)` to represent 0, though we want each "word" made up of the constructors of the type to represent a unique integer.
In our file, we describe how to define -3 with the above definition.

We may now define the integer that we think of when subtracting two natural numbers:

Subtraction of naturals to integers

```
def subNatNat (m n : ℕ) : int :=
match (n - m : Nat) with
| 0        => int.ofNat (m - n)
| (Nat.succ k) => int.negSucc k
```

This definition (adapted from mathlib) is a bit tricky. To subtract m from n, we look at the natural number $n-m$. If it's 0, then $n \leqslant m$ (try to prove this), so the subtraction of m by n should be the integer representing the natural number $m-n$. If $n-m$ isn't 0, in which case it must have form $k+1$ for some natural number k, then $n > m$, so that $m-n$, which we expect to be $-(n-m) = -(k+1) = -k-1$, will be represented with the help of `int.negSucc`.

In our file, we include the definition of addition of integers. Take note of the syntax to pattern-match on two inputs simultaneously. You may want to try to define the negative of an integer and subtractions of integers, as an exercise. Then, *ctrl + click* on the mathlib versions to the latter definitions we added in our file, to get a solution.

Finally, we discuss typecasting/coercion for the first but not last time in this tutorial.
Consider the infoview for the following lines:

Naturals to integers

```
#check 2
#check (2 : ℤ)
#check -2

#check Int.add
#eval Int.add 2 (-3)
```

When writing 2, Lean will consider it as a natural number by default. With syntax $(2 : \mathbb{Z})$, we tell Lean to consider it as an integer. What is happening here is that $(2 : \mathbb{Z})$ is read as `int.ofNat 2`. For -2, Lean will consider it as an integer by default. What is (almost) happening here is that -2 is read as `Int.neg (Int.ofNat 2)`.

Now, consider the last two lines of the above code. `Int.add` requires integer inputs, but in the second line, Lean doesn't complain that we've given a natural number as first input. Here, Lean is smart enough to understand that it should put a `Int.ofNat` on the first input, so that it becomes an integer. If you define your own structures, in which there is some sort of embedding, Lean will not automatically handle these sort of abuses of types, and you have to typecast explicitly. There'll be more on this in the chapter on type-classes.

To close this section, we mention that most concepts defined for natural numbers have counterparts in the integers. In our file, we provide the example of quotients and remainders for integers. Take note of the notation for these two, which is the same as the standard notation for $\mathbb{N}$.

2.4 Divisibility

We'll now briefly discuss divisibility, primes and the greatest common divisor, though the main aim of this section is to give you some more context for Lean and mathlib in general.

For example, we'll introduce divisibility in action:

Divisibility, and definitional equality

```
example (n m : ℤ)  : n ∣ m ↔ ∃ k, m = n*k := by rfl
```

First, watch out: this isn't your keyboards "|", it's a different one (yes ...), obtained by typing a backslash followed by a "|" (remember to hover above expressions to see how their typed in).

Now, the reason we may close the proof with `rfl` (which stands for **refl**exivity), is because the left expression of the equivalence is defined to be the right expression. If you swap the terms in the product of the right side expression, the proof will fail ! Often, for a given definition, there is a mathlib lemma that ends in `_def` that will replace an object with its definition, when used with `rw`. In this example, it's `dvd_def`.

Sometimes, you may want to write `_iff_` and press *ctrl + space* to make the auto-complete appear, hoping that you'll find your desired characterization among its suggestions. In our file, we give the example of the definition of coprime numbers, which are defined in terms of the gcd, and for which `Nat.coprime_iff_gcd_eq_one` is the desired characterization.

This "definitional equality" (the community uses the alias "defeq") has interesting side effects:

Definitional equality in action

```
example (n m : ℤ)  : n ∣ m → m % n = 0 :=
  by
  intro h
  obtain ⟨k,eq⟩ := h
  rw [eq]
  exact Int.mul_emod_right n k

example (n m : ℤ)  : n ∣ m → m % n = 0 :=
  by
  rintro ⟨k,eq⟩
  rw [eq]
  exact Int.mul_emod_right n k
```

In the first example, using the `obtain` line should theoretically raise an error, as `h` isn't of the pattern that `obtain` may be used on. However, if we unfold the definition of `h : n ∣ m`, which is `h : ∃ k, m=n*k`, we see that `obtain` works with the latter. What's happening is that `obtain` first unfolds definitions, so that the first example works. In the second example, we give a command specifically designed to introduce objects packaged in a quantifier or constructor to the proof state, `rintro`, where the `r` stands for **r**ecursive.

The latter example's statement seems pretty elementary: is it implemented in mathlib ? You may use commands `apply?` or `rw?` (they may take a while to complete), to have Lean display you possible implications or equivalences (respectively), that work on, and sometimes even solve your current goal.

apply? and rw?

```
example (n m : ℤ)  : n ∣ m → m % n = 0 :=
  by
  apply Int.emod_eq_zero_of_dvd

example (n m : ℤ)  : n ∣ m ↔ m % n = 0 :=
  by
  rw [@EuclideanDomain.mod_eq_zero]
```

These two examples can be solved with `apply?` or `rw?` respectively.

The theorem used in the proof in the last example mentions Euclidean domains. Generally, mathlib starts at fully abstract versions of concepts, which are then specialized to particular cases. Another example of this phenomenon is the property of a number of being prime, `Nat.Prime`, which is defined via irreducible elements of a monoid, though we often prefer to think of them as in the characterization `Nat.prime_def_lt''` we've included in the file (note the `def` in the name).

Next up in our exposition is the greatest common divisor, `Nat.gcd`. It is defined via Euclid's algorithm, so that we may compute actual gcd's, as is `#eval Nat.gcd 6 8`. We could also have expressed the latter using infix notation, as in `(6 : ℕ).gcd 8`, where we typecast so that Lean doesn't get confused. Often, no typecast is necessary:

Primes in infix notation, and `norm_num`

```
example (n : ℕ) (h : n = 7) : n.Prime := by rw [h] ; norm_num
```

The latter example shows that some properties may have infix notation too. Note that the fact that `norm_num` proves that 7 is prime is quite impressive: it requires a primality testing algorithm that is either verified for correctness, or present in a metaprogram, which is a program that writes Lean proofs.

We can't let you leave this section without giving you some practice. In our file, we've included to exercises for you to solve. In both we include the theorems you may use to solve them, presented in arbitrary order, at the end of the exercises in the `checks`. In the first exercise, you may want to use `apply?`. In the second exercise, to to prove the remaining goal without using `linarith`.

To conclude this section, we point you to the beginning of the file, where you will find:

Imports

```
import Mathlib.Data.Nat.Prime
import Mathlib.Data.Nat.GCD.Basic
```

You may *ctrl + click* on these paths to arrive at the cited mathlib files. Though we will discuss tools for finding the theorems one looks for when attempting a proof in the chapter on periferal content and its subsections, for now, we mention that exploring mathlib by looking at its files is the last resort option to learning a field in Lean, if no documentation is available online. As a cheap exercise, we recommend you explore these files to see what's at disposal for working with primes and gcd's. To the left in VS code, using the "Explorer" icon, you may access other mathlib files.

You'll notice in these files that there is a certain logic to the names of the theorems, which relates them to their statements. We present some of this nomenclature:

- **Nat** and **Int**, for naturals and integers respectively.
- **add** for addition, **sub** for subtraction, **mul** for multiplication, **div** for quotients, **mod** for remainders, `neg` for the negative.
- **le** for $\leqslant$ and **lt** for $<$, `eq` for $=$ and `neq` or `ne` for $\neq$.
- **iff** for equivalences, **of** for implications, `or` for disjunctions and `not` or `neg` for negation.
- Order of the names informs on the theorem. For example, `zero_add` is for $0 + a = a$ and `add_zero` is for $a + 0 = a$. Sometimes, directions are included in the name in the form of **left** or **right**.
- **comm** or **symm** for lemmata about exchanging parameters, `assoc` for lemmata about associativity of an operation, `trans` for lemmata about transitivity of an operation, ...

We will list nomenclature in each chapter and again in a dedicated chapter.
The nomenclature for this chapter would be:

- **dvd** for divisibility
- **prime**, **coprime** and **gcd** for their respective objects.

With this nomenclature, you can guess theorem names without having to look for them in mathlib files or using search engines. By starting the name of a theorem, you can press *ctrl + space* for VS Code to offer you auto-complete options that you may scroll through to find available lemmata.

3 Lists, multisets, and finite sets

3.1 Lists

Lists have two main aspects in Lean: they are used as data-structures over which one may perform algorithms, and they serve as foundations to the notion of a finite set. A Lean list is in other terms a "stack": there is the empty stack, and one may place objects on top of the stack to get a new stack. Formally:

Lists

```
inductive list (α : Type u) where
| nil : list α
| cons (head : α) (tail : list α) : list α
```

We require that all entries have the same type α. With the above definition, we're actually defining an entire family of types, as each `list` α is considered different for different α. The first constructor `nil` is a constant representing the empty list. Next, we have a constructor `cons` that tales a value and a list, and represents the list where this value was placed on top of the other list. In our file, we build `my_list`, which represents the list $(1, 2, 3)$. In standard Lean, lists may be defined with square bracket notation.
Contrary to lists in other languages, we may not access the entries of a list via an index (at least, not without some complications or conventions). Such an access would have to be a function from $\mathbb{N}$ to α, where each index in $\mathbb{N}$ would have to return some value of α, even if the index is larger then the lists length or if the list is empty.

We can however easily define the length of a list:

List length

```
def list.length : list α → ℕ
| list.nil => 0
| list.cons x l => 1 + list.length l
```

When reducing `list.length my_list` as we do in the file, Lean considers the last applied constructor used to build the input list: if it's `cons`, we have one more item, else, we've arrived at the empty list which we associate length 0. So the lengths are computed along the scheme $l(1, 2, 3) = 1+l(2, 3) = 1+1+l(3) = 1+1+1+l(0) = 3$, for example.

We may also define operations on lists, such as concatenation of lists:

List concatenation

```
def list.concat : list α → list α → list α
| list.nil , l => l
| list.cons x l , L => list.cons x (list.concat l L)
```

Here, we simply place the top entry of the first list on top of the concatenation of its tail with the second list. Concatenation is may be "computed" (reduced), along the scheme $(1, 2) + +(3, 4) = 1 :: ((2) + +(3, 4)) = 1 :: (2 :: (() + +(3, 4))) = 1 :: (2 :: (3, 4)) = (1, 2, 3, 4)$ where `++` represents concatenation and `::` represents `List.cons`, as they do in standard Lean.

The amazing thins is that we now may prove facts about lists and the operations on them! In the following for example, we prove that the length of a concatenation is the sum of the length of its terms.

List concatenation length

```
example (l s: list ℕ) : (list.concat l s).length = l.length + s.length :=
  by
  induction' l with x l ih
  · dsimp [list.length, list.concat]
    rw [zero_add]
  · dsimp [list.length, list.concat]
    rw [ih]
    rw [add_assoc]
```

It may seem weird to see an `induction'` appear in the proof. Yet, recall that this is a principle valid for all inductive types. If a property is true for the types constants, and is maintained when applying constructors, it's true for all terms of the type (because all terms are built that way, with a finite number of constructors).
In the latter proof, in the bases case, l is an empty list, and we may apply the corresponding case for empty lists for both `list.length` and `list.concat`, so that the goal simplifies to the trivial `s.length = 0 + s.length`. In the proof step, we check if the property is true when we extend a list with the unique constructor `cons`. This time, we are the the constructor-case of `list.length` and `list.concat`. This case quickly follow from the induction assumption. You may want to split the `dsimp`'s in this proof to get an appreciation of what is simplified in what order. Then, you may want to fill out the first `sorry` of our file as exercise.

Next, we may define properties and relations of lists in a manner similar to how we defined them for natural numbers. For example, we have:

List membership

```
inductive list.mem (x : α) : list α → Prop
| head (l : list α) : list.mem x (list.cons x l)
| tail (y : α) {l : list α} : list.mem x l → list.mem x (list.cons y l)
```

The idea behind the definition is that for a list of form `a :: l`, saying that `x ∈ a :: l` is equivalent to saying that `x=a`, which is encoded by the `head` constructor, or `x ∈ l`, which is encoded by the `tail` constructor.
As we exemplify in our file, showing that $2 \in (1, 2, 3)$ is achieved by `apply`ing the second constructor to have the result follow from $2 \in (2, 3)$ (since we have $(1, 2, 3) = 1 :: (2, 3)$ so that we may use $y = 1$ and $l = (2, 3)$), which in turn we have following from $2 = 2$ by by `apply`ing the first constructor (with $l = (3)$).

Another example of a property of lists ()of natural numbers) is that of being sorted increasingly:

Increasingly sorted list

```
inductive list.sorted : list ℕ → Prop
| empty_case : list.sorted list.nil
| cons_case :
      (∀ y : ℕ, list.mem y l → x ≤ y) → (list.sorted l) → list.sorted (list.cons x l)
```

To show that a list is sorted, we `apply` the second constructor until the list is empty, in which case we `apply` the first constructor. The second constructor asks us to show that all elements of the tail are greater then the head of the list, and that the list is sorted. The use of an implication to represent a conjunction ("and"), that we see here, is known as "*currying*", and you'll see it all over the place, as its technically useful.

In our file, we give three syntaxes for showing that $(1, 2, 3)$ is sorted. They each differ in the way case disjunction is handled. Each disjunct is on statements of the form `x ∈ a :: l` where the cases are `x=a` (for the `head` constructor) and `x ∈ l` (for the `tail` constructor). We'll let you explore these syntaxes by pointing you to the doc-strings and letting you follow the proof states in the infoview. It may be possible that future Lean versions will allow for a simpler syntax for these case disjunctions (that was present in Lean 3).
Take note of the `contradiction` tactic. What happens here is that the tactic notices that we've assumed that there's an element in an empty list. Yet, this property can't be built with the constructors of membership, as each such constructor returns the membership property for a list of form `a :: l`, a pattern that the empty list doesn't match. Thus, we have a false claim in our assumption, and by the *exfalso* principle, any statement follows.

In mathlib, the property of a list being sorted is actually an alias for a list's entries satisfying a relation pairwise. The way that this is defined is almost like our definition, but with an arbitrary relation instead of $\leqslant$. In our file, we give an example of a list of pairwise distinct entries. The latter fact is prove with `decide`, which will produce the tedious type of proof we gave for lists being sorted, fully automatically.

Finally, we ask you to fill out the second `sorry` of the file as exercise. As a hint: `norm_num` can prove contradictions, in a certain sense, as we illustrate in the `example` below. An explanation for what's happening in the latter is that the left side of the inequality has form `Nat.succ _`, while the right side is `Nat.zero`. Since Lean's foundations consider differently constructed (reduced) terms different, we've found our contradiction.

3.2 Algorithms on Lists

We'll now implement the insertion-sort algorithm for sorting lists of natural number in increasing order. Our main algorithm will recursively sort lists: given a list of form `x :: L`, we recursively sort `L`, and then we insert `x` into it before the first entry that is larger then `x`. We will then implement the following:

Correctness of insertion-sort
The list returned by insertion-sort is sorted

Proof: We'll reason by induction on the length of the list. Empty lists are sorted, so the base case holds. Now, given a list of form `x :: L`, since `L` alone has smaller size, we obtain from the induction hypothesis that insertion-sort will sort it. To see that the insertion procedure maintains the property of being sorted, note that inserting `x` doesn't affect the relation between the other entries, and that if `x` is inserted between a smaller entry and a larger one, since the list was sorted, it will be larger then all previous entries and smaller then all future entries □

First, we'll implement our insertion procedure:

Insertion procedure

```
def my_insert (x : ℕ) : List ℕ → List ℕ
| [] => [x]
| y :: l =>
      if x ≤ y
      then x :: (y :: l)
      else y :: (my_insert x l)
```

If `x` is smaller then the head-entry of the initial list, we pre-pend it to the list, otherwise, we insert it in the tail and pre-pend the head-entry.

Now, the insertion-sort algorithm can be stated quick promptly:

Insertion-sort algorithm

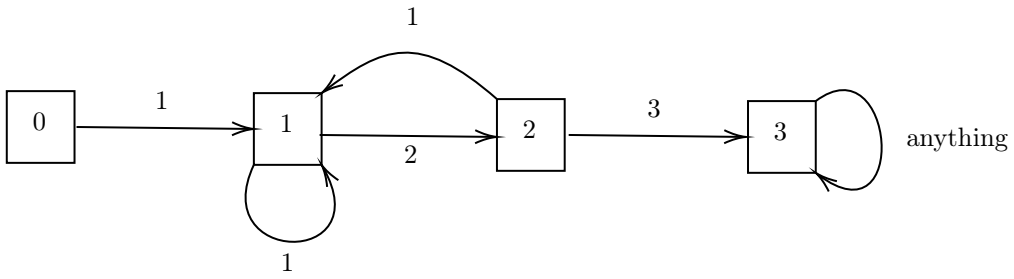
```
def insertion_sort : List ℕ → List ℕ
| [] => []
| x :: l => my_insert x (insertion_sort l)
```

We'll draw your attention to the following fact. As we show in our file, the algorithm is computable. This assumes that Lean can decide the validity of $x \leqslant y$ in `my_insert`. Indeed, Lean will produce proof, with some of its automation, for statements such as $x \leqslant y$ or their negation.

The fascinating thing is that we may now prove properties of these algorithms in the same language that we implemented them in! Our first such result will be the lemma `mem_insertion` of our file, that we won't reproduce here, due to the size of its proof. The proof is relatively elementary, as we don't use any too advanced theorems, yet we recommend you step through it, as it's techniques will be useful for the next lemma's proof, which we've given as exercise. Please fill in the sorry of `insertion_maintains_sort` using the lemmata we provide you in the `check`s below it, and with the lemmata used in `mem_insertion`.

The proof of correctness of insertion-sort is the `example` in our file. Just like the lemmata that came before it, no new techniques are used here, and you may consider it as a lengthier illustration of the things we introduced last chapter. Now, we'd like to introduce you to the concept of "*refactoring*": it' the process of breaking down proofs (or more generally programs, in the world of software development) into more elementary parts that may be reused in different proofs. This is supposed to avoid people from copy-pasting parts of proofs in the best case, and re-proving an implicitly existing lemma in the worst case, over and over again. You may want to think how our previous proofs may be refactored. Then, you may compare our exposition to that of file `Mathlib.Data.List.Sort` that you may reach by *ctrl + click*ing the corresponding `import`, where insertion-sort is implemented.

There are some amazing things we may do with lists. For example, consider the following automaton:



It's task will be to take as input a list of numbers and tell us if the sequence $1, 2, 3$ occurred in its entries. In the above diagram, boxes will indicate a state that we'll keep track of, and the arrows will correspond to the change of state that occurs when an entry of the number corresponding to that arrow is encountered. All arrows not displayed on the diagram will lead to state 0
The initial state is state 0. If a 1 is read, we move to state 1, symbolizing that we encountered a 1, otherwise, we stay at state 0. In state 1, we we move to state 2, symbolizing that we encountered sequence $1, 2$, if we read a 2. If we read a 1, the pattern is broken, but we recover the first number in the desired sequence, so that we move to state 1 in such a case. In state 2, we return to state 1 if we read a 1, for the same reason, and we move to state 3 symbolizing that pattern $(1, 2, 3)$ was encountered, if we read a 3. Reading anything else will return us to state 0. Once in state 3, we stay there no matter what is read, so that we may deduce that pattern $(1, 2, 3)$ has occurred, after the input list has been fully read.

We may implement such automata in Lean with the following operation:

Folding

```
List.foldl.{u, v} {α : Type u} {β : Type v} (f : α → β → α) (init : α) (l : List β) : α
```

Here α can be interpreted as the type of states of the automaton, and β the type of action-instructions that are sequentially read by the automaton. Then `f` will map a current state and a current action-instructions to a state that the automaton moves to. We start at initial state `init`, and read the actions from list `l`. The output of the operation will be the last state the automaton was in.

With this operation, we implement the automaton in our file. Note that the pattern appears in the second example of execution of the automaton, and we indeed get state 3 returned, which symbolized that the pattern has occurred. If you're looking for a challenge, you can try to formulate the property of a list of having pattern the sequence $1, 2, 3$ in it (perhaps with the `++` (`List.append`) operation ?), and formally prove our claim about the automaton.

We'll now give some nomenclature for lists, before moving on to sets:

- `mem` for membership. This is also the nomenclature for sets.
- `cons` and `nil` for the list constructors

3.3 Multisets

There is not a lot of interesting math implemented for finite multisets: they are mostly an intermediate step in the foundation of finite sets. However, this section will introduce you to a new way of forming types that is used in many other places in mathlib, so you better not skip it.

Our journey start with the property of two lists of being permutations of one another:

Permutation of lists

```
inductive perm : List α → List α → Prop
| nil : perm [] []
| cons (x : α) {l₁ l₂ : List α} : perm l₁ l₂ → perm (x :: l₁) (x :: l₂)
| swap (x y : α) (l : List α) : perm (y :: x :: l) (x :: y :: l)
| trans {l₁ l₂ l₃ : List α} : perm l₁ l₂ → perm l₂ l₃ → perm l₁ l₃
```

We hope you've gotten used to these sorts of definitions. Below, they are seen in action, where the $\sim$ (tilde) is infix notation for the relation, which requires the `open` line, as this symbol may appear in different context, so that we must specify this context.

Permutation of lists, exemplified

```
open List in
example : [1,2,3] ~ [1,3,2] :=
  by
  apply Perm.cons
  apply Perm.swap
```

There is even an algorithm computing all permutations of a list: `List.permutations`

The idea will be to define (finite) multisets to be lists "*up to*" permutation, as we consider elements of sets to be unordered, when considering equality of sets. This will be implemented by using a new way to construct types: quotients. They are based on equivalence relations, which we now take a closer look at.

In our file, we define the relation on the integers of having same remainder mod 3, and call it `my_rel`. We could prove that this relation is an equivalence relation as a theorem, and we do show that the relation is transitive in separate lemma our `as_lemma`, but Lean takes a different route. For properties that are bundles of object or properties, such as in our case, as the property of being an equivalence relation is a bundle of the relation being reflexive, symmetric and transitive, Lean offers us the possibility to store these facts in a *structure-type*. There will be more on this in the next chapter.
If we stored these properties in conjunctions, as in:

Don't try this at home, kids

```
example {α : Type u} {r : α → α → Prop} : Prop :=
  (∀ (x : α), r x x) ∧ (∀ {x y : α}, r x y → r y x) ∧
  (∀ {x y z : α}, r x y → r y z → r x z)
```

We'd constantly have to navigate them with `.left` and `.right` suffixes whenever we need to mention one of the constituent properties. Instead, with the syntax used in `my_equiv`, which we'll elaborate on in the next chapter, we store these properties in a structure, which allows us to refer to them by name. For example, transitivity of `my_rel` may be accessed by `my_equiv.trans`, whereas we'd have to refer to it with suffix `.right.right` using the above definition.

Long story short: as mathlib lemma `List.Perm.eqv` certifies, permutations on lists form an equivalence relation. Moving ahead in our file, we'll simply ignore the two `Setoid` related paragraphs that follow, to arrive at the notion of `Quotient`. Given an equivalence relation on a type, its `Quotient` may be though of as set type of equivalence classes. So for our example of having same remainder mod 3, we've obtained the type corresponding $\mathbb{Z}/3\mathbb{Z}$ in mainstream math (Watch out! This is not mathlib's definition of $\mathbb{Z}/3\mathbb{Z}$, which is derived quite differently). For the quotient type, we can make elements by wrapping the elements of the original type in the double-brackets we use in our file: think of this object as the equivalence class of the element. To see these concepts in action, we give two ways of showing that $\bar{2} = \bar{5}$ in $\mathbb{Z}/3\mathbb{Z}$ (we use regular brackets here, since we haven't figured out how to get the double ones):

$\bar{2} = \bar{5}$ **in** $\mathbb{Z}/3\mathbb{Z}$

```
example : ([2] : Quotient my_inst) = [5] :=
  by
  apply Quotient.sound
  dsimp [HasEquiv.Equiv, instHasEquiv, Setoid.r, my_inst, my_rel]
  decide

example : ([2] : Quotient my_inst) = [5] :=
  by
  apply Quotient.sound
  rfl
```

In the second example, `rfl` unfolds definitions and performs computations/reductions in one swoop.

Similarly, multisets are the quotient of lists for the permutation-relation. In our file, we show how to prove that two multisets are equal. Now, without going into the details of quotients, we'll give some examples of how operations and properties on lists can be "*lifted*" to multisets.

Lifting as a concept fundamental to quotients. If we have a quotient on type α given by relation r, and we consider a function f that takes the same value for all elements of a same equivalence class under r, then we may define a similar function for the equivalence classes themselves. Formally, we require that $\forall x, y,\ r\ x\ y \Rightarrow f(x) = f(y)$. The lifted f will then be defined for all elements of the quotient. Its value for an input equivalence class is the value of any of its representatives, since f takes the same value on the latter.

We may lift any function, including predicates, so that we may lift membership of lists:

Lifting membership

```
example (x : ℕ) (m : Multiset ℕ) : Prop :=
Quot.lift
   (List.Mem x)
   (by
    intro a b arb
    dsimp [Setoid.r] at arb
    apply propext
    exact List.Perm.mem_iff arb
   )
   (m)
```

Indeed, as `List.Perm.mem_iff` states, for any list representative of the class representing a multiset, if an element x is in that list, then it is in all permutations of that list, i.e. it is in all representatives of the class representing a multiset.

Next, we'll lift an operation on lists, `List.union`, which will later on serve as foundation to the union of finite sets. It's based on the operation `List.insert`, which is also good to know of.

More lifting

```
example (m M : Multiset ℕ) : Multiset ℕ :=
Quot.lift₂
   (fun l L => [List.union l L])
   (by
    intro a b c brc
    rw [Quotient.eq]
    apply List.Perm.union
    · exact List.perm_rfl
    · exact brc
   )
   (by
    intro a b c arb
    rw [Quotient.eq]
    apply List.Perm.union
    · exact arb
    · exact List.perm_rfl
   )
   (m) (M)
```

Take note of the fact that the lifted property is, in a sense, decidable (we could show it with `decide`), and the lifted operation is evaluable. Indeed, the only legal way to build elements of a quotient type is by taking a element of the initial type, and wrapping it in a constructor (`Quot.mk`) to get it equivalence class. This means that when given an element of the quotient type, we may always extract an element of the initial type, for which we can compute the value or decide the property, which will then be the value of the lifted function or hold for the quotient element. However, taking a representative of a quotient element may not be a computable Lean function. In our `my_inst` example, `Quot.sound` implies that `Quot.mk my_rel 2 = Quot.mk my_rel 5`. If our hypothetical representative function existed, applying it to each side would yield $2 = 5$ in $\mathbb{Z}$. So the `eval` must do computation outside the logical framework of Lean (or at least that's what I'm guessing, I'm no expert).
Rest assured: this discussion is not essential for you to understand at this stage.

We conclude our very brief discussion of multisets and quotients by asking you to fill in the sorry at the end of our file as exercise, in order to lift the `List.append` operation to multisets. To succeed, you might need a lemma similar to `List.Perm.union`: we what tell you waht it is, but instead ask you to guess it using nomenclature.
For the sake of completeness, we lift addition from $\mathbb{Z}$ to our $\mathbb{Z}/3\mathbb{Z}$ in the last code-block of our file.

3.4 Finite sets

Finally, we're back to an object encountered more frequently in mainstream math: finite sets. Finite sets are a finite unordered collection of pairwise distinct objects. With multisets, we satisfy the first two properties, so we'll consider the last one. Again, our journey will start at lists. In our file, you may *ctrl + click* on `List.Nodup`, to see that it's based on `List.Pairwise`, which we encountered when defining the property of lists being sorted. We may lift this property to `Multiset.Nodup`, since all permutations of a list that has distinct elements will also have distinct elements. Now, we may define finite sets as:

Finite sets

```
structure finset (α : Type) where
   val : Multiset α
   nodup : Multiset.Nodup val
```

They are defined as a structure-type that bundles the actual object we'll consider to be the finite set, `val`, which is just a multiset, together with the property of this multiset having pairwise distinct elements, `nodup`. Bundling up these two things may seem strange, but they allow `(f : Finset α)` to replace the pair `(f : Multiset α) (hf : Multiset.Nodup f)`, whenever we define an finite set when stating a lemma/theorem. To our great pleasure, structures allow us to refer to their entries with the use of suffixes, as we'll exemplify in our file, right after defining building the finite set $\{1, 2, 3\}$ in our framework, named `my_finset`. Take note of the way that object of a type built as a structure are defined: we use `where` instead of `:=`, and then we name the entries of the structures, and fill them with objects of their requested type. If the entry was a property, we must provide a proof.

A finite set

```
def my_finset : finset ℕ where
   val := {1,2,3}
   nodup :=
      by
      --decide -- Works! But let's take a walk:
      rw [Multiset.nodup_iff_pairwise]
      dsimp [Multiset.Pairwise]
      use [1,2,3]
      constructor
      · rfl
      · decide
```

To see this formalism in action, we'll define a operation on finite sets:

Union of finite sets

```
def my_union (s t : finset ℕ) : finset ℕ where
  val := Multiset.ndunion s.val t.val
  nodup :=
    by
    apply Multiset.Nodup.ndunion
    exact t.nodup
```

The actual operation on the multiset representing the finite set will be `Multiset.ndunion`, which we encountered last section already. The reason why we preferred this notion of union to that obtained by lifting append now becomes clear: to get an operation on finsets, we must certify that `Multiset.ndunion` maintains the property of having pairwise distinct elements (which isn't the case when lifting `List.append`). The mathlib lemma `Multiset.Nodup.ndunion` certifies this fact. Take note of the fact that in the above proof, the fact that `t.val` has distinct elements, `t.nodup`, isn't displayed, as it's information contained in `t : finset α`. Structure types therefore make the proof state in the infoview a bit less informative: you'll have to get used to this.

Similarly to unions, the length of a list can be lifted to mutlisets to provide `Multiset.card`, the size of a multiset. Then to define the size of a finite set, `Finset.card`, we simply take the size of its `val` entry.
This notion of finiteness we've developed so far is completely unrelated to the mainstream math way of defining finiteness. The latter consists of defining the set of the first n natural numbers, and saying that a set has size n if we can put it in bijection (a one-to-one correspondence) with that set.
We actually have this characterization in mathlib:

Finiteness

```
Finset.card_congr
  {α : Type} {β : Type} {s : Finset α} {t : Finset β}
  (f : (a : α) → a ∈ s → β)
  (h₁ : ∀ (a : α) (ha : a ∈ s), f a ha ∈ t)
  (h₂ : ∀ (a b : α) (ha : a ∈ s) (hb : b ∈ s), f a ha = f b hb → a = b)
  (h₃ : ∀ b ∈ t, ∃ a, ∃ (ha : a ∈ s), f a ha = b)
  : Finset.card s = Finset.card t
```

The bijection is `f`. Since functions are between types, we model restraining `f` to the set `s` by requiring a certificate that the first input is in the set, as second input. Take good note of this, as it's a common trick used to model restrained functions in mathlib, though alternatives (subtypes) exist, which we'll soon explore. Similarly, to model the fact that `f` has range `t`, we use assumption $\texttt{h}_1$. Assumptions $\texttt{h}_2$ and $\texttt{h}_3$ are injectivity and surjectivity of the function, respectively.

You may wonder how this lemma is proved, so we'll sketch the proof. We first show that t is the image of s under f, `Finset.image`. Wen enter the computationally flavored world, based on `Finset.image_val_of_injOn`, which relates images to the map operation (for example, look at the algorithm `List.map`). Injectivity is then related to the map operation maintaining the property of pairwise distinctness, using `Multiset.Nodup.map_on`.

Now, enough with the foundations! Let's look at finite sets in action. In our file, we give a sequence of examples of relations and operations on finsets: subsets, intersection, set difference, the empty set, and disjoint unions. We've added their nomenclature at the end of the chapter.

A particular operation on finsets is filtering, which consists in taking a finite set S and a property $P(x)$ that elements x may or may not have, and creating $\{x \in S | P(x)\}$. This operation is `Finset.filter` in mathlib. It has a certain sublety to in on which we now elaborate.

As you can see from the two first application examples of our file, filtering allows us to compute the prime numbers of a finset of natural numbers, or to compute the numbers satisfying some special equation. However, this isn't possible for all properties we might want to filter on: we give two examples of this in our file. The predicate in the latter example is related to Fermat's last theorem, so we must expect failure, at least until Kevin Buzzard is done with formalizing FLT. This is where we encounter the difference between classical and constructive mathematics. In classical math, we may define such finite sets, even if we're unable to compute them, whereas this is prohibited in constructive math. Since mathlib tries to be as general as possible, and since constructive math is valid in the classical context, but the converse doesn't necessarily hold, we must use `open Classical in` before definitions, lemmata, commands, that will make use of classical math. If you desire using classical math for the rest of your file (or section/namespace, which we'll discuss later on), you may also just write `open Classical`.

Alternatively, you can try to stay in constructive math, and show that the predicate you're filtering on is decidable, like this, for example:

Example of decidability

```
instance : DecidablePred (fun x : ℕ => ∃ y, y ∣ x) :=
  by
  intro x
  apply isTrue
  dsimp
  use x
```

A predicate is decidable if we can show, without classical math of course, that it is true or false. The next example of decidability we give in our file is a little more insightful (but a bit longer) then the previous one. Before discussing it, we note that once decidability is shown, we may `eval` the finset filtered from this property.

In our second example of decidability, we wish to show that for a given natural number x, it's decidable that there exists a divisor y of x that is strictly smaller then x. This may be true of false, depending on x: for $x \in \{0, 1\}$, it's false (for natural numbers), and for other values it's true, as we may use divisor 1. The point is that we may exactly say when it is true or when it is false. For each value of x, we can point of a proof of truth or falsity for that value x.

We'll discuss decidability in more depth in a later chapter. For now, we'll note that decidability may lead to curious errors when working with finite sets. For example, Lean will complain if we write:

```
(fun (α : Type) (a : α) (s : Finset α) => s.filter (fun x => x = a))
```

The reason is that equality is a property, and there are - a priori - types for which equality hasn't been shown to be decidable. Below in our file, we show how to use `DecidableEq` to get decidable equality as an additional assumption on the type. This concludes our first discussion on decidability, and on filtering.

Next, up we exemplify how to work with explicitly constructed finite sets. Similarly to `cons` for lists, the finset $\{1, 2, 3\}$ is interpreted as `insert` 1 $\{2, 3\}$. We see how this plays out in action in an example in our file. We then give a second version of the proof, using the `fin_cases` tactic, which does the unpacking for us.

Finally, we'll take a look at some standard finite sets, and some more operations on finite sets:

- `Finset.range` represent the sets $\{0, 1, ..., n\}$.
- The `Finset.I`-family represent intervals. The `c` stands for closed and `o` for open, and they determine whether the provided bounds are included in the interval or not.
- `Finset.powerset` returns the finite set of all subsets of a set.
- `Finset.powersetCard` returns the finite set of all subsets of a set of a prescribed size. This is the same as `Finset.slice`. Mathlib has and will probably continue to have multiple versions of the same concept, which is a fact one must get used to.

Now, isn't it time for an exercise ?
Seeing as many low-level properties surrounding finite sets are present in mathlib, we have given you as exercise one of them, but ask you to provide a different proof then by simply applying the mathlib lemma. You may try to fill in the `sorry` using the two lemmata we give as `check`s below. Also, take note of the `ext` command, which in this context alows you to prove that two sets are equal if the elements of one are in the other.

Finally, to close this section, we address questions you may have had throughout the section.
Do we have general, possibly infinite sets in Lean ? Why are finite set not defined from these general sets ?
To answer the first question: we do have the `Set` type (more accurately, type constructor). We don't really need sets in type theory, but seeing as may mathematical subjects are framed in the language of sets, they were added to mathlib, and are used in these subjects (for example, point-set topology). However, using sets to "define" elements, as we usually do in math, is not recommended in Lean. The reason for this is that Lean's sets are actually just predicates in disguise. For example, if the set $\{1, 2, 3\}$ can be associated to the predicate on $\mathbb{N}$ that states $x \mapsto x = 1 \vee x = 2 \vee x = 3$. Membership corresponds to satisfying the predicate, intersections correspond to taking conjunctions of predicates, subsets correspond to predicates implying each other, equality corresponds to equivalence of predicates, and so on. Thus, statements about sets are just rephrasing about properties.
We can map finsets s to sets by defining set $\{x | x \in s\}$, where the predicate is finset membership, which is multiset membership, which was lifted from list membership. Conversely, we may define a notion of finiteness for `Set`, `Set.Finite`, which relates them to finsets. It goes a little something like this:

Finiteness of sets

```
structure set.finite {α : Type } (s : Set α) where
  elems : Finset α
  prop : ∀ x, x ∈ s → x ∈ elems
```

Thus, a set is finite precisely when we may provide an finset `elems` and a proof that all elements of the set are in `elems`. The actual mathlib definition is based on one more concept of finiteness, which we'll investigat in the chapter on typeclasses. One of these notions isn't actually based on finsets, but follows the more mainstream version we previously mentioned These notions are equivalent (and this is implemented), though the one that's most wildly used is the one based on finsets.
This leads us to answer the second question we previously asked. Finiteness seems to have been derived from lists as it makes them computable, in a sense. We could have defined finiteness by first defining the `Sets` $\{n : \mathbb{N} | n < m\}$ first and saying that a set is finite if it's in bijection with the latter. However, the framework of lists was already present, and making this alternative definition computation friendly (how to `eval` a union of finite sets, for example) is - though a priori possible - not the way things went.

Here's the promised nomenclature:

- `Finset` for finite sets, `Set` for sets, `empty` for $\emptyset$
- `subset` for $\subseteq$ and `ssubset` for $\subset$ (s for strict), `filter` for subsets built with filtering.
- `union` for $\cup$, `inter` for $\cap$, `sdiff` for set difference $\setminus$.

3.5 Enumeration

TODO

3.6 Big operators

With finite sets defined, we may now define another fundamental mathematical tool: finite sums and products. There is nice notation for them, and they are evaluable:

```
#eval Σ x in {1,2,3}, x
```

We'll discuss sums, but products behave very similarly. Working with sums and products requires you to open the `BigOperator` namespace.

Sums are defined in terms of the fold operation which we described in the chapter on algorithms on lists, where we interpreted it as an automaton. Think of sums as an automaton that starts at state 0 and moves to states by taking the numbers to be summed as action, where each new sate is the addition of that number with the previous state.

In our file, we see sums in action in this nice result:

Summing like Gauss

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof: We don't have to sum in a specific order, so if we sum the first and last, then the second a previous to last number and so on, we note that we're always getting the intermediate term $n+1$. To give a cleaner version of the idea, lets write the sum twice, once in increasing term order and once in reverse order, and add its term pairwise in order:

$$\underbrace{\begin{matrix} 1 & 2 & \dots & n \\ n & n-1 & \dots & 1 \end{matrix}}_{n\times(n+1)}$$

We see that we get n additions always returning $n(n+1)$. Since this is twice the same sum, the result follows. □

Proofs like this are nice for humans, but less nice for formal verification. Inverting summation order and showing that the pairwise additions are all $n+1$ is - though possible - quite involved in terms of formalism. This is why we will formalize the following second proof:
Proof: We show this by induction on n. In the base case, both sides are 0, so that the equality holds. Assuming that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ and noting that $\frac{(n+1)(n+2)}{2} = \frac{n(n+1)}{2} + (n+1)$ we immediately get $\sum_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}$, so that the step holds. □

In our file we formalize this proof, for the following formalization of the statement:

```
example (n : ℕ): Σ  m in Finset.range (n+1), m = (n*(n+1))/2 :=
```

Alternatively, we could have used `Finset.Icc 1 n` to describe $\{1, 2, ..., n\}$. This seemingly minor difference is actually quite significant from the formal perspective. If you need this theorem in an `Finset.Icc 1 n`, you have to first translate the latter result into this context. Recall that `Finset.range (n+1)` is actually $\{0, 1, 2, ..., n\}$ so you'll need more work then showing a simple set equality. This is of course doable, but time-consuming. Another aspect to the choice of formalization is the following: proving the theorem with `Finset.Icc 1 n` might be harder, if less API is provided for `Icc` then for `range`. This discussion highlights that the choice of formalism is quite impactful and requires some thought, as well as some mathlib cultivatedness.

As exercise, we ask you to fill out the first `sorry` of our file, in which you'll show that the sum of even numbers is even. We've provided you the necessary mathlib lemmata as disordered `check`s below. As a bonus exercise, show the same result, but with one of `Icc` or `Ico` instead of `range`. Note any differences ?

Next, we'd like you to have a look at the following phenomenon:

A bug ?

```
noncomputable
def my_func (n : ℕ) (hn : n ≠ 1) :=
  Classical.choose (Nat.exists_prime_and_dvd hn)

example : 1 ≤ Σ x in Finset.Icc 2 5, my_func x (_) := by sorry
```

The scary looking function `my_func` doesn't have to be complicated on purpose. You may replace it with:

```
def my_func (n : ℕ) (hn : n ≠ 1) : ℕ := n
```

---

and the same phenomenon will occur. The reason we use the latter is because this is a function that, morally speaking, is a partial function. To see what that means, we'll say what it does: it returns a prime divisor of an input n. However, not all natural numbers have prime divisors: indeed, 1 has only itself as divisor, but isn't prime. As we saw for finiteness via bijections, we may fromalise such "partial" functions, which aren't defined for all inputs of the input type, by adding the restricting property as second input. So here, we need input `hn`.

And here comes the twist. When writing the statement of the above `example`, note that at the underscore, Lean expects a proof that the summand is indeed distinct of 1. This shouldn't be a problem, as we're summing over $\{2, 3, 4, 5\}$. Yet, if we consider the infoview at this spot, Lean seems to have forgotten that the summand is in $\{2, 3, 4, 5\}$! What's going on here ?
If you hover above `Finset.sum`, you will note that the function who's image we'll take the sum over (the identity, for simply summing the terms of set), has type $\alpha \to \beta$. So we aren't in the framework of partial functions, which require a second input. We mentioned that sums are defined via the fold operation. To be able to sum over function images, we use the map operation before folding. The map operation doesn't support our sort of partial functions either. It is actually an interesting question whether it is possible to define a map operation that supports partial functions. For example, for the list version of the map operation, which is defined as:

`List.map`

```
def map (f : α → β) : List α → List β
| []    => []
| a::as => f a :: map f as
```

How would we define this operation for $(f : (a : \alpha) \to p\ a \to \beta)$ for some predicate p ?

An idea for a sort of workaround for this problem is the following: if we bundle a value and the proof that it satisfies the property into a single type, we can use this type to define partial functions, and simultaneously use the framework for the previously mentioned operations. This is what leads us to "*subtypes*" which may be defined as:

Subtypes

```
structure subtype (α : Type) (p : α → Prop) where
   val : α
   prop : p val

#check {n : ℕ // n.Prime}
```

The `check` is an example for the notation one may use to define subtypes, where $\alpha = \mathbb{N}$ and p is `Nat.Prime`.

For our problem, we may use `Finset.attach`, which will produce a finset of the subtype for the property of belonging to the original finset. To see how it works in practice, consider:

Subtypes

```
example : 1 ≤ Σ x in Finset.attach (Finset.Icc 2 5),
   my_func x (by
             have h := x.prop
             rw [Finset.mem_Icc] at h
             linarith
             ) :=
```

For an `x` of a subtype property of the value `x.val` is accessible via `x.prop`.

TODO: transition, or go directly ? ask on Zulip about "pprod" API.

4 Types and terms

4.1 Types as foundations

4.2 Types as information-carriers

4.3 Propositions as types (!)

4.4 Recursion

4.5 Quotients

4.6 Decidability

5 Tactics

5.1 What are tactics ?

5.2 Common tactics

5.3 Applied meta-programming

repeat try and all that

6 Classes

6.1 Basics on type classes

The `#synth` to access proof of instance

6.2 Common type classes

6.3 Solutions

7 Combinatorics

7.1 The pigeonhole principle

7.2 Double counting

7.3 Solutions

8 Graphs

8.1 Basics on graphs

8.2 Reiman's theorem

9 Linear (leanear) Algebra

10 Geometry

10.1 Sylvester-Gallai

11 Probability

12 Algorithms

13 Category theory

13.1 Basics

13.2 First-fit bin-packing

14 Peripheral content

14.1 Workflow

14.2 Tools

moogle, aesop and AI

14.3 Common mistakes

15 Miscellaneous

15.1 Starting a project

In the terminal, move to the directory in which you would like to start the project and run
`lake +leanprover/lean4:nightly-2023-02-04 new NAME math` where *NAME* is the name of the new folder that will store the project. Next, move to the new *NAME* folder, and run `lake update` followed by
`lake exe cache get`. It's advised to keep your project files that will contain your code in a new folder inside the project-folder. Make a new file filled with the following, to see if things worked out:

```
import Mathlib.Topology.Basic

#check TopologicalSpace
```

15.2 Lean tool in VS code

In VS code, in between the "run code" and "split editor" icons, you'll find a ∀ icon that allows you to access Lean project management tools for example. You may use it to start projects, download projects, restart/reload files, update dependencies including mathlib, etc. Alternatively, enter the "command palette" via "View", or press *ctrl* + *p*, and start typing `> Lean 4 :`. The auto-complete will suggest Lean management tools.
Important things are:

- **Updating mathlib**: access it via "Project Actions" or via the command palette: "Project : Update Dependency...". Then, select mathlib. To track the progress of the update, consult the bottom bar in VS code.
-

15.3 elan

Elan is Leans version management tool. You may run it from the terminal/shell of your operating system, from any directory. Some important commands are :

- `elan help` will display the available commands with short descriptions
- `elan show` will display the current version of Lean you're using
- `elan default` *version-name* will set the Lean version you're using to *version-name*.
- `elan update` updates Lean and elan

15.4 lake

15.5 Lean code highlighting in LaTeX