# 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 $n$th 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.

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. 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

## 2.2 Euclidean division

## 2.3 Basics on integers

## 2.4 Divisibility

## 2.5 Solutions

# 3 Lists, multisets, and finite sets

## 3.1 Basics on Lists

## 3.2 Algorithms on Lists

## 3.3 Multisets

## 3.4 Finite sets

## 3.5 Enumeration

## 3.6 Solutions

# 4 Types and terms

## 4.1 Types as foundations

## 4.2 Types as information-carriers

## 4.3 Sets, functions and relations

# 5 Tactics

## 5.1 What are tactics ?

## 5.2 Common tactics

## 5.3 Applied meta-programming

repeat try and all that

### 5.4 Solutions

## 6 Classes

### 6.1 Basics on type classes

### 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 Peripheral content

### 11.1 Workflow

### 11.2 Tools

moogle and AI

### 11.3 Common mistakes

## 12 Miscellaneous

### 12.1 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

### 12.2 lake

### 12.3 Starting a project

### 12.4 Lean code highlighting in LaTeX