A Gentle Introduction on Lean

Moti Ben-Ari

http://www.weizmann.ac.il/sci-tea/benari/

Version 0.5

January 4, 2024

Contents

1 Introduction

Lean 4 is a *proof assistant*. You enter your proof into Lean in a formal language and the system checks the correctness of the proof. It displays the current set of hypotheses and goals, and it is capable of performing many simple proofs automatically. This tutorial is, as its name states, a gentle introduction to Lean intended for students and others who have no previous experience with a proof assistant.

Complete Lean proofs of theorems of arithmetic and logic are given. The source code is heavily commented and additional explanations are given. As you gain more experience with Lean, the number of comments on a proof is reduced. Proof tactics are introduced one at a time as needed as are tips. Tables of keyboard shortcuts, tactics and tips are given in the appendices. The tips include information on definitions and notations used by Lean.

This tutorial limits itself to proving properties of integers and natural numbers, for example, theorems on greatest common denominators and prime numbers. Beyond that you need a basic knowledge of propositional and first-order logic as found in introductory textbooks.[1]

The Lean source code of these proofs and the XeLaTeX source of this document can be found at

`https://github.com/motib/gentle-lean`.

Installation

To install Lean, follow the instructions for your operating system at

`https://leanprover-community.github.io/get_started.html`.

The Lean community uses *Visual Studio Code (VSC)* `https://code.visualstudio.com/`. It is a very versatile environment with lots of features, so be sure to study VSC tutorials before starting to work with Lean.

You *must* work within a project framework as described in

`https://leanprover-community.github.io/install/project.html`.

When you start Lean to work on an existing project, you must open the *Folder* containing the project. Once you have created a project, you can create new source files which must have the extension `lean`.

Tips for working with Lean

Tip: Infoview

Enter `ctrl-shift-enter` to open the Lean Infoview where hypotheses, goals and errors are displayed. To understand the effect of applying a tactic, I have found it helpful to place the cursor just before the source line, and then to alternate between `Home` and `End` while looking at the Infoview.

Tip: tactic

If you hover over a tactic its specification will be displayed.

For example, the specification of the tactic `exact` is

`exact e` closes the main goal if its target type matches that of `e`.

[1]Only tactics are used in the proofs and they are applied one at a time.

Tip: theorem

You can display the statement of a theorem by hovering over its name.

Theorems are defined to be widely applicable, so initially you will not be able to understand the full statement of a theorem. In this tutorial only natural numbers and integers are used, so you can interpret the theorems narrowly, for example,

```
    dvd_mul_right (a b : Nat) : a ∣ a * b
```

Preamble of a Lean program

The Lean community has developed an extremely large mathematical library called `Mathlib`. You will work with the definitions and theorems defined there, which must be imported as the first lines in you Lean source file.

```
    import Mathlib.Tactic
```

imports the proof tactics.

You will need least one other library that contains the definitions and theorems of the data type that you want to work on, for example,

```
    import Mathlib.Data.Nat.Prime
```

The examples here are enclosed in

```
    namespace gentle
```

to ensure that their names do not clash with the names of theorems in `MathLib`. In addition,

```
    open Nat
```

will often be used to open the namespace of `Nat` so that we can write `factorial` instead of `Nat.factorial`.

Preamble do not appear in the examples here but can be found in the Lean source files.

Syntax

Lean uses two types of comments:

- Line comments start with `--` and continue to end of the line.
- Range comments start with `/-` and continue to `-/`.

Within Mathlib the convention is to use spaces around each operator:

```
    a ≤ b ∧ b ≤ a → a = b
```

and I will do so in this tutorial. You may ignore this convention in your proofs: `a≤b∧b≤a→a=b`.

There is also a convention for naming theorems: the name is written in lower case with underscores between the parts of the name. The name specifies the meaning of the theorem in a few words or abbreviations, for example, `min_le_right` is `min a b ≤ b` and `min_le_left` is `min a b ≤ a`.

References

Mathematical logic

- M. Ben-Ari. *Mathematical Logic for Computer Science (Third Edition*, Springer, 2012.
  An introduction to mathematical logic including both syntactical and semantic proof methods, as well as sections on temporal logic and program verification.

- M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems (Second Edition)*, Cambridge University Press, 2004.
  An introduction to mathematical logic with emphasis on natural deduction, temporal logic and model checking.

Lean resources

- J. Avigad and P. Massot. *Mathematics in Lean*.
  `https://leanprover-community.github.io/mathematics_in_lean/`
  This is a comprehensive presentation of the use of Lean to prove mathematical theorems. The proofs in this document are based on examples given there.

- J. Avigad, L. de Moura, S. Kong and S. Ullrich. *Theorem Proving in Lean 4*.
  `https://lean-lang.org/theorem_proving_in_lean4/`
  A formal presentation of Lean.

- The website of the Lean Community contains links to important resources and to the Zulip chat where you can ask questions.
  `https://leanprover-community.github.io/`

Learning materials on Lean

- The games on the Lean Game Server are a fun way to start learning Lean.
  `https://adam.math.hhu.de/`

- D. J. Velleman. *How to Prove It with Lean*.
  `https://djvelleman.github.io/HTPIwL/`
  This work focuses on proof techniques in mathematics; to simplify learning it uses tactics developed by the author.

- H. Macbeth. *The Mechanics of Proof*.
  `https://hrmacbeth.github.io/math2001/`
  This work focuses on proof techniques in mathematics; to simplify learning it uses tactics developed by the author.

Acknowledgment

I am indebted to the members of the Lean Community for their patience and help as I took my first steps in Lean.

2 Commutativity of the greatest common denominator

A theorem is declared like a function in a programming language with the name of the theorem, its parameters and the statement of the theorem. The keyword `by` introduces a *tactic proof*, which we will be using exclusively in this tutorial.

```
1 /-
2   Commutativity of gcd
3 -/
4 theorem gcd_comm (m n : ℕ) :
5     Nat.gcd m n = Nat.gcd n m := by
```

The parameters `m n` which are declared as natural numbers have a logical meaning as universally quantified variables.

$$\forall m, n \in N : (\text{gcd}(m, n) = \text{gcd}(n, m)) .$$

The proof of the theorem uses the fact that the division operator is antisymmetric: if $m|n$ and $m|n$ then $m = n$. Applying this theorem results in two new goals.

```
6   apply Nat.dvd_antisymm
7     -- dvd_antisymm: (m ∣ n ∧ n ∣ m) → m = n,
8     --   where m = gcd m n, n = gcd n m
9     -- Two new goals: gcd m n ∣ gcd n m, gcd n m ∣ gcd m n
```

When you place the cursor before line 6, Lean Inforview will display the *tactic state*:

```
1 goal
m n : ℕ
⊢ Nat.gcd m n = Nat.gcd n m
```

This gives the *context* that `m n` are natural numbers and the current *goal* following the turnstile symbol ⊢. After the theorem `Nat.dvd_antisymm` is applied, the tactic state changes to

```
2 goals
case a
m n : ℕ
⊢ Nat.gcd m n ∣ Nat.gcd n m
case a
m n : ℕ
⊢ Nat.gcd n m ∣ Nat.gcd m n
```

By the anti-symmetry of division, you can prove an equality by proving that the two sides divide each other, thus creating two goals.

Tactic: `apply`

If you have a theorem `P→Q` and the goal matches[a] `Q` then `apply ⊢P→Q` removes the goal `Q` and adds `P` as the new goal.

If you have a theorem `Q` and the goal matches `Q` then `apply Q` removes the goal and there are no more goals.

[a]Technically, if the goal *unifies* with `Q`, but we don't explain the concept here.

gcd(m, n) will divide gcd(n, m) only if it divides both n and m, so we again have two new subgoals. Then, by definition, gcd(m, n) is a common divisor so it divides both the left parameter m and the right parameter n.

```
  -- First goal
  apply Nat.dvd_gcd
      -- dvd_gcd: (k ∣ m ∧ k ∣ n) → k ∣ gcd m n,
      --   where k = gcd m n, m = n, n = m
      -- New goals are gcd m n ∣ n and gcd m n ∣ m
  apply Nat.gcd_dvd_right
      -- gcd_dvd_right: gcd m n ∣ n,
      --   where m = m, n = n
  apply Nat.gcd_dvd_left
      -- gcd_dvd_left: gcd m n ∣ m
      --   where m = m, n = n
```

Repeat the same proof for the second subgoal gcd$(n, m) \mid$ gcd(m, n).

```
  -- Second goal
  apply Nat.dvd_gcd
      -- dvd_gcd: k ∣ m ∧ k ∣ n → k ∣ gcd m n,
      --   where k = gcd n m, m = m, n = n
      -- New goals are gcd n m ∣ m and gcd n m ∣ n
  apply Nat.gcd_dvd_right
      -- gcd_dvd_right: gcd m n ∣ n,
      --   where m = n, n = m
  apply Nat.gcd_dvd_left
      -- gcd_dvd_left: gcd m n ∣ m
      --   where m = n, n = m
  done
```

Tip: `done`

All proofs should be terminated by `done`. This is not necessary but if your proof is not complete, `done` will display a message.

Tip: Division

The division operator in Lean is not the `|` symbol on your keyboard, but a similar Unicode symbol `∣` obtained by typing `\|`.

3 Commutativity of minimum

To prove that the minimum operator is commutative, a hypothesis named `h` is introduced. Of course we have to prove the hypothesis before we can use it.

```
/-
  Commutativity of minimun
-/
theorem min_comm (a b : ℕ) :
    min a b = min b a := by

  have h : ∀ x y : ℕ, min x y ≤ min y x := by
    -- Hypothesis: For all natural numbers x y,
    --   min x y ≤ min y x
```

Tactic: `have`

Introduces a new named hypothesis for use in the proof.

Tip: indentation

All statements used to prove a hypothesis must be indented a fixed number of spaces.

The hypothesis is $\forall m, n \in N : (\min(m, n) = \min(n, m))$. For the bound variables m, n, free variables must be substituted. The tactic `intro x y` introduces new free variables `x y` for `m n`.

```
    intro x y
      -- Introduce arbitrary x y in place of ∀
```

Tactic: `intro`

Introduces free variables in place of bound variables in a universally quantified formula. A universally bound variable means that the formula has to hold for an arbitrary value so we substitute a variable that is this arbitrary value.

If the goal is is `⊢P→Q`, `intro` introduces `P` as a hypothesis. If `Q` is proven then the hypothesis can be *discharged* and `P→Q` is proven.

The first step of the proof of the hypothesis uses the theorem that $c \leq a \wedge c \leq b \rightarrow c \leq \min(a, b)$. The Lean theorem `le_min` is applied to obtain two subgoals whose proofs complete the proof of the hypothsis `h`.

```
    apply le_min
      -- le_min: for all natural numbers c,
      --   (c ≤ a ∧ c ≤ b) → c ≤ min a b
      --   where a = y, b = x, c = min x y
      --   New goals are min x y ≤ y and min x y ≤ x
    apply min_le_right
      -- min_le_right: min a b ≤ b
      --   where a = x, b = y
      -- Solves goal min x y ≤ y
    apply min_le_left
      -- min_le_left: min a b ≤ a
      --   where a = y, b = x
    -- This completes the proof of h
```

Now that h has been proved, the indentation is removed to continue the proof the main theorem. le_antisymm is applied to the goal creating two subgoals which are proved by applying the hypothesis h.

```
  apply le_antisymm
    -- le_antisymm: (a ≤ b ∧ b ≤ a) → a = b
    --   where a = min a b, b = min b a
    --   New goals are min a b ≤ min b a and min b a ≤ min a b
  apply h
    -- Apply h with x = a and y = b
  apply h
    -- Apply h with x = b and y = a
  done
```

4 Cancellation properties

Here we prove theorems of arithmetic related to cancellation properties:

$$-a + (a + b) = b\,, \quad a + c = a + c \rightarrow b = c\,, \quad a * 0 = 0.$$

The library `import Mathlib.Data.Int.Basic` is imported because negative numbers are integers, not natural numbers.

Tip: associativity

The associativity of the operators `+` and `*` is defined to be *left*:
`a + b + c` is `(a + b) + c` and `a * b * c` is `(a * b) * c`.
Use `add_assoc` and `mul_assoc` to change the associativity of an expression.

In order to prove $-a + (a + b)$ we first need to proof that it equals $(-a + a) + b)$ and then set $-a + a = 0$.

This proof uses the tactic `rw` that *rewrites* an expression with another expression that has already been proved to be equal to it. After two rewrites, applying the theorem `zero_add b` results in exactly the expression needed to proof the goal.

```
theorem neg_add_cancel_left (a b : Int) :
    -a + (a + b) = b := by
  rw [← add_assoc]
    -- add_assoc: a + b + c = a + (b + c)
    -- Addition is left associative: a + b + c = (a + b) + c
    -- New goal is -a + a + b = b
  rw [add_left_neg]
    -- add_left_neg: -a + a = 0
    -- New goal is 0 + b = b
  exact zero_add b
    -- zero_add: 0 + a = a, where a = b
  done
```

Tactic: `rw`

`rw [eqn]`, where `eqn` is an equation or an equivalence, rewrites the goal by replacing occurrences of the the left-hand side of `eqn` with the right-hand side.
`rw [← eqn]` rewrites the goal by replacing occurrences of the right-hand side of `eqn` with the left-hand side.
`rw [eqn] h` and `rw [← eqn] h` rewrite expressions in the hypothesis `h`.

The arrow ← which is defined as "rewrites the goal by replacing occurrences of the right-hand side of `eqn` with the left-hand side" can seem non-intuitive: when `← add_assoc` is used to rewrite the goal `-a + (a + b) = b` above, we are rewriting the *left-hand side* of the goal. However, the directions are referring to the sides of `eqn` (here, `a + b + c = a + (b + c)`) where occurrences of *its* right-hand side are `a + (b + c)` are replaced by *its* left-hand side `a + b + c`.

Tactic: `exact`

The tactic `exact h` is used when `h` is exactly the statement of the current goal, so the goal is now proved. `exact` is similar to `apply` but limited because it can only be used if the hypothesis exactly matches the goal.

The parameter `h : a + b = a + c` in the theorem `add_left_cancel` is a hypothesis that is used in the proof. It must be provided when the theorem is applied (see below).

```
theorem add_left_cancel {a b c : Int}
    (h : a + b = a + c) : b = c := by
  rw [← neg_add_cancel_left a b]
    -- neg_add_cancel (above): -a + (a + b) = b
    -- New goal is -a + (a + b) = c
  rw [h]
    -- New goal is -a + (a + c) = c
  exact neg_add_cancel_left a c
    -- neg_add_cancel (above): -a + (a + b) = b, where b = c
  done
```

The theorem `add_left_cancel` has four parameters: `a b c` of type `Int` and a hypothesis `h`. To use the theorem we can write

```
    apply add_left_cancel i j k h
```

for some values `i j k` of type `Int` and a hypothesis `h`. When the theorem is used in line 36 below, `h` has been replaced by its definition in line 25:

```
    exact add_left_cancel (a * 0 + a * 0 = a * 0 + 0)
```

but the integer parameters are not given. When the hypothesis is matched against `a + b = a + c` in the statement of the theorem, we have

```
    a = a * 0   b = a * 0   c = 0.
```

Clearly, all three expressions are of type `Int` so Lean can infer their values and types when they are used:

```
    exact add_left_cancel (a * 0) (a * 0) 0 h
```

Tip: implicit

Parameters declared with braces, such as `{a b c : Int}` state that `a b c` are *implicit* bound variables in the theorem whose actual names and types can be inferred when the theorem is used.

The proof of the following theorem is similar to the previous two. First the hypothesis `h` is proved using `rw` three times and it is then used as a hypothesis in the application of `add_left_cancel`.

```
theorem mul_zero {a : Int} :
    a * 0 = 0 := by
  have h : a * 0 + a * 0 = a * 0 + 0 := by
    rw [← mul_add]
      -- Distribute multiplcation over addition (reversed)
      -- mul_add: a * (b + c) = a * b + a * c,
      --   where a = a, b = 0, c = 0
      -- New goal is a * (0 + 0) = a * 0 + 0
    rw [add_zero]
      -- add_zero: a + 0 = 0
      -- New goal is a * 0 = a * 0 + 0
    rw [add_zero]
      -- h is proved
  exact add_left_cancel h
    -- add_left_cancel (above): if a + b = a + c then b = c,
    --   where a = a * 0, b = 0, c = 0
  done
```

5 There exists an infinite number of prime numbers

Assume to the contrary that there are finitely many prime numbers $p_1, p_2, \ldots, p_n$. Consider $q = (p_1 p_2 \cdots p_n) + 1$. For any $1 < m < q$, m does not divide q because the remainder is 1. Therefore, q is a prime since it is divisible only by 1 and q.

There are two ways to define primes in Lean:

- `Nat.prime_def_lt`: p is prime if and only if $(2 \leq p) \wedge \forall m < p(m|p \rightarrow m = 1)$.
- `Nat.Prime.eq_one_or_self_of_dvd`: p is prime only if $m|p \rightarrow (m = 1 \vee m = p)$.[2]

The proof above follows the definition `Nat.prime_def_lt`, but the proof using
`Nat.Prime.eq_one_or_self_of_dvd`
is easier, because we need only work the hypothesis $m|p$ and not with the quantifier $\forall m < p$.

The proof starts with the *definition* of p as the smallest prime factor of $n! + 1$. A major part of the proof is to show that such a p does exist.

```
theorem exists_infinite_primes (n : ℕ) :
    ∃ p, n ≤ p ∧ Nat.Prime p := by
      -- For all natural numbers n,
      --   there exists a natural number p,
      --   such that n ≤ p and p is a prime
  let p := minFac (n ! + 1)
    -- if n ! + 1 ≠ 1, p is its smallest prime factor
```

Tactic: `let`

`let` introduces a definition whose scope is local.

Now we prove the hypothesis that $n! + 1 \neq 1$.

```
  have f1 : n ! + 1 ≠ 1 := by
    apply Nat.ne_of_gt <| succ_lt_succ <| factorial_pos _
      -- factorial_pos: n ! > 0
      -- succ_lt_succ: m < n → succ m < succ n
      --   where m = 0, n = n !
      --   succ(essor of) n is the formal definition of n + 1
      -- ne_of_gt: b < a → b ≠ a
      --   where b = 1 and a = n ! + 1
      -- <| means that the formula on its right is
      --   the input to the one on its left
```

Let us look at `apply Nat.ne_of_gt <| succ_lt_succ <| factorial_pos _` in detail.

- `factorial_pos _` is the theorem that the value of any factorial is positive, here, $0 < n!$.

[2]This definition is *not* if and only if because $p = 1$ satisfies the right-hand side but 1 is not a prime.

- `succ_lt_succ` uses the definition of $+1$ as the successor function. It takes $0 < n!$ from `factorial_pos _` and proves that $0 + 1 < n! + 1$.
- `Nat.ne_of_gt` is the simple theorem that if $1 < n + 1!$ then $1 \neq n + 1!$, which is the hypothesis `f1` that we want to prove.

Tip: right-to-left

The symbol `<|` means to compute the expression to its right and pass it to the left.

From the hypothesis `f1` it follows that $n! + 1$ has a smallest prime factor and therefore p exists.

```
  have pp : Nat.Prime p := by
    apply minFac_prime f1
      -- minFac_prime: if n ≠ 1 then
      --   the smallest prime factor of n prime,
      --   where n = n ! + 1
      -- f1 proves pp
```

To prove $n \leq p$, we will prove the equivalent formula $\neg(n \geq p)$ which itself if equivalent to $(n \geq p) \rightarrow \mathit{False}$. The method is to introduce $n \geq p$ as a new hypothesis and derive a contradiction.

```
  have np : n ≤ p := by
    apply le_of_not_ge
      -- le_of_not_ge: ¬a ≥ b → a ≤ b
      --   where a = n, b = p
      -- New goal is ¬n > p
    intro h
      -- By definition of negation, n ≥ p implies False
      -- Assume n ≥ p and make False the new goal
      --   to prove np by contradiction
```

Tip: negation

A negation `¬p` is *defined* as `p → False`. The tactic `intro` makes `p` into a hypothesis and `False` into the goal. When there are no more goals, `¬p` has been proved (by contradiction).

On page 6 we noted that the tactic `intro` is applicable to goals or hypotheses of the form `P → Q`, as is done here with `p → False`.

Two final hypotheses are needed: $p|n!$ and $p|1$.

In the proof of the first hypothesis, the theorem `dvd_factorial` takes two parameters: the fact that the minimum prime factor of any number (_) is positive and the hypothesis `h` that `n ≥ p`.

```
    have h₁ : p ∣ n ! := by
      apply dvd_factorial (minFac_pos _) h
        -- minFac_pos: 0 < minFac n,
        --   where _ means that this holds for any n
        -- dvd_factorial: (0 < m ∨ m ≤ n) → m ∣ n !
        --   where m = p
        -- p is natural so 0 < m and
        -- p ≤ n by assumption (intro) h
        -- h₁ is proved
```

To prove the second hypothesis we use the theorem `Nat.dvd_add_iff_right` whose main operator is if-and-only-if $a \leftrightarrow b$. This theorem can be used in two ways:

- Modus ponens (`mp`): $a \rightarrow b$, assume a and prove b.
- Modus ponens reversed (`mpr`): $a \leftarrow b$, assume b and prove a.

Here the reversed modus ponens is used.

```
    have h₂ : p ∣ 1 := by
      apply (Nat.dvd_add_iff_right h₁).mpr (minFac_dvd _)
        -- minFac_dvd: minFac n ∣ n,
        --   where _ means that this holds for any n
        -- dvd_add_iff_right: k ∣ m → (k ∣ n ↔ (k ∣ m) + n)
        --   where k = p, m = n !, n = 1
        --   p ∣ n ! by h₁, so p | 1 iff p ∣ n ! + 1
        -- mpr (MP reverse): p ∣ n ! + 1 → p | 1
        --   p ∣ n ! + 1 by definition
        --   since pp shows that p is prime
        -- p ∣ is proved
```

Applications of the hypotheses `pp` and `h₂` complete the proof of `np`.

```
    apply Nat.Prime.not_dvd_one pp
      -- if p is a prime (true by pp) then ¬p ∣ 1
      --   which is p | 1 → False
    exact h₂
      -- Use MP with h₂, proving False and
      --   thus np by contradiction,
      --   since by definition a prime is ≥ 2
    -- The proof of np : np: n ≤ p is (finally) complete
```

The current goal is now $\exists p(n \leq p \wedge prime(p))$. The existentially quantified variable p must be replace with a specific value which is also p using the tactic `use`. This leaves the conjunction $n \leq p \wedge prime(p)$, which is split into two subgoals that are proved using the hypotheses `np` and `pp`.

```
  use p
    -- Introduce free variable p for the bound variable p
    --   to get n ≤ p ∧ Nat.Prime p
    -- Since both conjuncts are hypotheses,
    --   the proof is complete
  done
```

Tactic: `use`

Given an existential goal `∃ c A(c)`, this tactic introduces a free variable for the bound variable `c` to form a new goal. It will also attempt to prove the goal using the hypotheses.

Even without the comments the proof is not short. When proving complex theorems it is convenient to assume that certain lemmas are true, and then when the main proof is complete to return to prove the lemmas. In Lean the tactic `sorry` is accepted as a proof of anything. For example, the eight-line proof of the hypothesis `np` can be proved by `sorry`. Lean will issue a stern warning to tell you not to rest on your laurels.

```
theorem exists_infinite_primes (n : ℕ) :
    ∃ p, n ≤ p ∧ Nat.Prime p := by
  let p := minFac (n !  + 1)
  have f1 : n ! + 1 ≠ 1 := by ...
  have pp : Nat.Prime p := by ...
  have np : n ≤ p       := by sorry
  apply Exists.intro p
  apply And.intro
  · apply np
  · apply pp
  done
```

Tactic: `sorry`

Proves any theorem.

6 The square root of two is irrational

Suppose that $\sqrt{2}$ is rational so that $\sqrt{2} = m/n$. Without loss of generality, assume that m, n are *coprime*, that is, they have no common factor. Then $m^2/n^2 = 2$ and $m^2 = 2n^2$, so 2 must divide m. Therefore, 2^2 must divide $2n^2$ and hence n is also divisible by 2, contracting the assumption that m, n are coprime. This section contains a Lean proof of the central claim of the proof that if m, n are coprime then $m^2 \neq 2n^2$.

A `lemma` is just a different name for a `theorem`. This lemma proves that $a^2 = a \cdot a$ using `rw` on the successor function and the base case of the definition for taking the power of a number.

```
lemma pow_two (a : ℕ) : a ^ 2 = a * a := by
  rw [Nat.pow_succ]
    -- Nat.pow_succ:  n ^ succ m = n ^ m * n,
    --   where n = a, m = 1, succ m = 1 + 1
    -- New goal is a ^ 1 * a = a * a
  rw [pow_one]
    -- pow_one: a ^ 1 = a
  done
```

The second lemma proves that if $2|m^2$ then $2|m$ using the first lemma and the theorem that if there is a prime factor of $a * b$ then it is a prime factor of either a or b. When the hypothesis `h` which is now $2|m \cdot m$ is rewritten, since $a = b = m$ the hypothesis becomes the disjunction $2|m \vee 2|m$.

```
lemma even_of_even_sqr (m : ℕ)
     (h : 2 ∣ m ^ 2) : 2 ∣ m := by
  rw [pow_two] at h
    -- pow_two (lemma): a ^ 2 = a * a,
    --   where a = m
    -- New goal is 2 ∣ m
  rw [prime_two.dvd_mul] at h
    -- prime_two: 2 is prime
    -- dvd_mul: if p is prime
    --   then p ∣ m * m ↔ p ∣ m ∨ p ∣ m,
    --   where p = 2, m = m,
    -- Apply to h : 2 | m * m → (2 | m ∨ 2 | m)
    -- h is now 2 | m ∨ 2 | m, goal is still 2 ∣ m
```

Split the disjunctive hypothesis $2|m \vee 2|m$ into two identical hypotheses; each one is exactly the goal of the lemma.

```
  rcases h with h₁ | h₁
    -- Splits disjunctive hypothesis h:
    --   2 | m ∨ 2 ∣ m  into
    --   two (identical ) subformulas 2 ∣ m, 2 ∣ m
```

Prove both the (identical) subformulas.

```
  · exact h₁
      -- 2 ∣ m proves 2 ∣ m
  · exact h₁
      -- 2 ∣ m proves 2 ∣ m
  done
```

Tactic: `rcases`

Given a hypothesis or goal that is a disjunction `A∨B` the tactic `rcases` splits it into two sub-hypotheses or subgoals `A` and `B`.

For a disjunctive hypothesis, prove the goal under both sub-hypotheses. For a disjunctive goal, prove one of the subgoals.

To prove the theorem, we assume $m^2 = 2n^2$ and try to prove a contradiction.

```
theorem sqr_not_even (m n : ℕ) (coprime_mn : Coprime m n) :
    m ^ 2 ≠ 2 * n ^ 2 := by
  intro sqr_eq
    -- Assume sqr_eq: m ^ 2 = 2 * n ^ 2 and
    --   prove a contradiction
```

Add the hypothesis $2|m$. By the lemma `apply even_of_even_sqr`, it is sufficient to prove $2|m^2$ which becomes the new goal. Rewrite using the assumption $m^2 = 2n^2$ that was introduced to prove a contradiction and apply the trivial theorem that $a|a \cdot b$ proving the hypothesis.

```
  have two_dvd_m : 2 ∣ m := by
    apply even_of_even_sqr
      --  even_of_even_sqr (lemma): 2 ∣ m ^ 2 → 2 ∣ m
      --  New goal is 2 ∣ m ^ 2
    rw [sqr_eq]
      -- sqr_eq: m ^ 2 = 2 * n ^ 2.
      -- Apply to the current goal.
      -- The new goal is 2 ∣ 2 * n ^ 2
    apply dvd_mul_right
      -- dvd_mul_right: a ∣ a * b,
      --   where a = 2, b = n ^ 2
      -- Apply to the current goal to prove two_dvd_m : 2 ∣ m
```

The definition of divisibility is: $a|b \leftrightarrow \exists c(a \cdot c = b)$. In Lean this is expressed by the forward direction (`mp`) of `dvd_iff_exists_eq_mul_left` applied to the hypothesis `2 ∣ m`. The goal is now to find such a c.

```
  have h : ∃ c, m = c * 2 := by
    apply dvd_iff_exists_eq_mul_left.mp two_dvd_m
      -- dvd_iff_exists_eq_mul_left: a ∣ b ↔ ∃ c, b = c * a
      --   where a = 2, b = m, c = c
      -- Use MP with two_dvd_m: 2 ∣ m to prove h
```

Given an existential formula such as $\exists c P(c)$, let c be some value that satisfies P.

```
rcases h with ⟨k, meq⟩
  -- h : ∃ c, m = c * 2 is an existential formula
  --   rcases on h:
  --     k is the free variable for the bound variable c
  --     meq : m = k * 2 is a new hypothesis
  -- Type ⟨ ⟩ using \< \>
```

Tactic: `rcases`

`rcases h with ⟨v, h'⟩` means given a hypothesis `h`, let `v` be a value such that the new goal is `h'`.

By now you are certainly quite skillful in Lean, so the extent of the comments in the source code will be reduced.

We now prove a sequence of five hypotheses.

```
have h₁ : 2 * (2 * k ^ 2) = 2 * n ^ 2 := by
  rw [← sqr_eq]
    -- sqr_eq : m ^ 2 = 2 * n ^ 2
    -- ← is right to left rewriting of 2 * n ^ 2 in h₁
    -- New goal is 2 * (2 * k ^ 2) = m ^ 2
  rw [meq]
    -- Rewrite m = k * 2 in h₁
    -- New goal is  2 * (2 * k ^ 2) = (k * 2) ^ 2
  ring
    -- Prove goal by using the ring axioms
```

Tactic: `ring`

Proves equalities that can be proved directly from the axioms of a commutative ring without taking any hypotheses into account. For the ring of integers, only addition, subtraction, multiplication and powers by natural numbers can be used. The division operation is not defined in the ring of integers because $1/2$ is not an integer.

It is easy to see that $2(2k^2) = (k \cdot 2)^2$ can be proved using only the definition of positive powers of integers as repeated multiplication, and the laws of associativity and commutativity of the integers.

The theorem `mul_right_inj'` is applied assuming that $2 \neq 0$, but this is a simple property of natural numbers which can be proved by the tactic `norm_num`.

```
have h₂ : 2 * k ^ 2 = n ^ 2 := by
  apply (mul_right_inj' (by norm_num : 2 ≠ 0)).mp h₁
    -- mul_right_inj': a ≠ 0 → (a * b = a * c ↔ b = c)
    --   where a = 2, b = 2 * k ^ 2, c = n ^ 2
    -- norm_num: solves equalities and inequalities like 2 ≠ 0
    -- Since 2 ≠ 0, MP on h₁ proves h₂
```

Tactic: `norm_num`

Proves numerical equalities and inequalities that do not use variables.

```
  have h₃ : 2 ∣ n := by
    apply even_of_even_sqr
      -- even_of_even_sqr (lemma) : 2 ∣ m ^ 2 → 2 ∣ m
      --   where m = n
      -- New goal is 2 ∣ n ^ 2
    rw [← h₂]
      -- Rewrite right-to-left of h₂ in the goal
      -- New goal is 2 ∣ 2 * k ^ 2
    apply dvd_mul_right
      -- dvd_mul_right : a ∣ a * b,
      --   where a = 2, b = k ^ 2 to prove h₃
```

```
    have h₄ : 2 ∣ Nat.gcd m n := by
      apply Nat.dvd_gcd
        -- Nat.dvd_gcd : (k ∣ m ∧ k ∣ m) → k ∣ gcd m n
        --   k = 2, m = m, n = n
        -- New goals are 2 ∣ m and 2 ∣ n
      · exact two_dvd_m
        -- First goal is two_dvd_m
      . exact h₃
        -- Second goal is h₃
```

```
  have h₅ : 2 ∣ 1 := by
    rw [Coprime.gcd_eq_one] at h₄
      -- if m and n are coprime then gcd m n = 1,
      --   where m = 2 and n = 1
      -- Apply to h₄
      -- New goals are 2 ∣ 1 and m, n are coprime
    exact h₄
      -- Proves 2 ∣ 1
    exact coprime_mn
      -- Assumption that m, n are coprime
```

The sequence of hypotheses that have been proved terminates in `h₅ : 2 ∣ 1`, but `norm_num` can prove that this is the negation of the true formula `1 ∣ 2`, thereby deriving a contradiction.

```
  norm_num at h₅
    -- Goal is 2 ∣ 1
    -- norn_num can prove that this is False
    -- Proving the contradiction of the initial assumption
  done
```

7 Propositional logic

This section demonstrates tactics that can be used to prove theorems that use the operators of propositional logic. The final subsection presents the tactic for tautologies, which can immediately complete some proofs.

7.1 Conjunction and equivalence

When a theorem has equivalence (⊚) as the main operator and it must be split into two subgoals, one for modus ponens (mp) direction and one for the modus ponens reversed (mpr) direction.

```
theorem lt_iff_le_eq {a b : Int} :
    a < b ↔ a ≤ b ∧ a ≠ b := by
  rw [lt_iff_le_not_le]
    -- lt_iff_le_not_le : a < b ↔ (a ≤ b ∧ ¬b ≤ a)
  constructor
    -- Create two subgoals (mp and mpr) from the current iff goal
    -- First subgoal is (a ≤ b ∧ ¬b ≤ a) → (a ≤ b ∧ a ≠ b)
```

Tactic: `constructor`

Splits a goal into two subgoals: equivalence (↔) into two implications (Modus ponens and Modus ponens reversed) and conjunction (∧) into two conjuncts.

The current goal is an implication whose premise and conclusion are both conjunctions. We can use `intro` to introduce the premise as a hypothesis and then `rcases` (page 16) to split the conjunctive hypothesis into two subgoals to be proved.

```
intro h
rcases h with ⟨h0, h1⟩
```

It is possible to combine the two tactics into the tactic `rintro`.

```
  -- Prove the mp goal
  · rintro ⟨h0, h1⟩
```

We start with the implication of the mp direction, where `rintro` introduces the hypothesis and then `constructor` splits the conjuctive goal into two subgoals.

```
  · rintro ⟨h0, h1⟩
    -- rintro introduces the premise as a hypothesis and also
    --   performs an rcases on the hypothesis to split it
    --   two sub-hypotheses a ≤ b and ¬b ≤ a
    constructor
      -- Creates two subgoals from the current conjunctive goal
    · exact h0
        -- Proves the second subgoal
    · intro h2
        -- a ≠ b is a = b → False
      apply h1
        -- Replace False with the negation of the hypothesis
      rw [h2]
        -- Proof is complete since b ≤ b
```

The proof of the implication of the mpr direction is similar.

```
  · rintro ⟨h0, h1⟩
    constructor
    · exact h0
    · intro h2
      apply h1
      apply le_antisymm h0 h2
        --  le_antisymm: (a ≤ b) → (b ≤ a) → a = b
  done
```

Tactic: `rintro`
Performs `intro` and then `rcases` to split the resulting hypothesis.

7.2 Disjunction

We now prove a theorem with an equivalence and a disjunction. In the previous theorem, we first split the equivalence into two implications. Here, tactic `rcases` is used with a theorem to split on the sign of y.

```
theorem lt_abs {x y : Int} :
    x < |y| ↔ x < y ∨ x < -y := by
  rcases le_or_gt 0 y with h | h
    -- Absolute value depends on sign of y
    -- le_or_gt 0 y: a ≤ b ∨ a > b
```

The equivalence goal is unchanged, but we are tasked with proving it under both hypotheses: $0 \leq y$ and $0 > y$. First we prove for $0 \leq y$, in which case we can rewrite $|y|$ by y.

```
  · rw [abs_of_nonneg h]
    -- abs_of_nonneg: 0 ≤ a → |a| = a
    constructor
      -- Split iff into mp and mpr
    · intro h'
```

In the mp formula, the premise is introduced as a hypothesis and the conclusion becomes the goal $x < y \vee x < -y$. When a *goal* is a disjunction, it is sufficient to prove one disjunct. (Of course, if the *hypothesis* is a disjunction, we have to prove the theorem for *both* possibilities.) Here we are smart enough to tell Lean that we want to prove the left disjunct, because the right one won't make any progress toward the proof.

```
      left
        -- The current goal is a disjunction
        --   so tell Lean which disjunct we want to prove
      exact h'
    . intro h'
      rcases h' with h' | h'
        -- The hypothesis is a disjunction and we have to prove
        --   the goal for each disjunct
      · exact h'
      . linarith
        -- The hypotheses are 0 ≤ y and x < -y
        -- Lean can prove that this implies the goal x < y
```

Tactic: `left`, `right`

If the goal is a disjunction, tell Lean which disjunction you want to prove.

Tactic: `linarith`

The tactic solves linear equalities and inequalities. Unlike `ring` it can use hypotheses and unlike `norm_num` it can solve equations with variables.

The proof for $y < 0$ is similar.

```
  · rw [abs_of_neg h]
    constructor
    · intro h'
      right
      exact h'
    . intro h'
      rcases h' with h' | h'
      · linarith
      . exact h'
  done
```

7.3 Implication

The following theorem is proved by contradiction using the tactic `by_contra`, after which the proof is straightforward.

```
theorem T1a {A : Prop} : (¬A → A) → A := by
  intro h1
  by_contra h2
    -- Prove A by contradction: assume A and prove False
  apply h2
    -- Modus ponens
  apply h1
    -- Replace goal by the hypothesis
  exact h2
  done
```

Tactic: `by_contra`

This tactic removes a goal `P`, adds the hypothesis `¬P` and creates a new goal `False`.

Here is another proof of the same theorem, this time using contraposition instead of contradiction. The result of using the tactic `contrapose` will be unfamiliar. Given the *hypothesis* $\neg A \rightarrow A$, it does not change the hypothesis into $\neg A \rightarrow \neg\neg A$. Instead, the hypothesis becomes the goal $\neg(\neg A \rightarrow A)$ and the goal A becomes the hypothesis $\neg A$.

This makes sense by the deduction theorem.

$$\{H_1, H_2, \ldots, H_n\} \vdash G$$

means

$$\vdash H_1 \wedge H_2 \wedge \ldots \wedge H_n \rightarrow G$$

whose contrapositive is

$$\vdash \neg G \rightarrow \neg(H_1 \wedge H_2 \wedge \ldots \wedge H_n)\,,$$

which using deduction is

$$\neg G \vdash \neg(H_1 \wedge H_2 \wedge \ldots \wedge H_n)\,.$$

Although $\neg(\neg A \rightarrow A)$ doesn't simplify the proof, if we push negation inward (do it yourself`!`) the result is $\neg A \wedge \neg A$ which is trivial to prove. The exclamation point in the tactic means that following the tactic `contrapose`, the tactic `push_neg` is called to push negation inward.

```
theorem T1b {A : Prop} : (¬A → A) → A := by
  intro h1
  contrapose! h1
    -- Replace h1 by its contrapositive
    -- Push negation inward (!)
  constructor
    -- Split conjunction
  · exact h1
  · exact h1
  done
```

Tactic: `contrapose`

Transforms a goal into its contrapositive. Applied to a hypothesis, it makes the negation of the goal into a hypothesis and the negation of the hypothesis into the goal. An exclamation point following `contrapose` calls tactic `push_neg` on the resulting contrapositive.

The interesting step in the following proof is

```
    rcases h1 with ⟨h2, _⟩
```

The hypothesis `h1` is `A ∧ ¬B` while the goal is `A`. We use `rcases` to split the hypothesis into two, `A` and `¬B`. Since only one sub-hypothesis is sufficient to prove the goal, we don't even bother to give the sub-hypothesis `¬B` a name.

```
theorem T2 {A B : Prop} : ((A → B) → A) → A := by
  intro h1
  contrapose! h1
    -- Replace h1 by its contrapositive
    --   and push negation inward
  constructor
    -- Split conjunction
  · contrapose! h1
      -- Goal is A → B, h1 is ¬A
      -- Make ¬(A → B) the hypothesis and A the goal
      -- Push negation inward (!)
    rcases h1 with ⟨h2, _⟩
      -- Split the hypothesis
    · exact h2
      -- Only need to use left subformula
      --   of the conjunction
  · exact h1
  done
```

Tip: Don't care

When a name or value is syntactically required but you don't care what its value is, you can use the underscore symbol `(_)` instead.

The next theorem we want to prove $(A \to B) \vee (B \to C)$ has a disjunction operator as its main operator. We prefer to carry out the proof using only the implication operator, so we first try to prove the hypthosis $\neg(A \to B) \to (B \to C)$.

When the hypothesis has been proved, use the `contrapose!` to negate the hypothesis and the goal and then exchang them. Moving the negations inward results in a hypothesis that is exactly the goal.

```
theorem T3 {A B C : Prop} : (A → B) ∨ (B → C) := by
  have h1 : ¬(A → B) → (B → C) := by
    -- Prove the implication equivalent to the disjunction
    intro h2
    intro h3
    contrapose! h2
      -- Contrapositive of ¬(A → B)
    intro
      -- No need to name the new hypothesis
    exact h3
  contrapose! h1
  exact h1
  done
```

7.4 Tautologies

Tautologies in propositional logic can be proved very easily using semantic methods such as truth tables and semantic tableaux. Lean can prove tautologies so the above theorems can be proved immediately using the tactic `tauto`.

```
theorem T1a {A : Prop} : (¬A → A) → A := by tauto
theorem T2  {A B : Prop} : ((A → B) → A) → A  := by tauto
theorem T3  {A B C : Prop} : (A → B) ∨ (B → C) := by tauto
```

Tactic: `tauto`
If a tautology can be formed from the hypotheses and the goal, the proof can be immediately completed using this tactic.

The following theorem that `tauto` works when the tautology is formed from both a hypothesis and a goal.
`theorem T0a {A : Prop} (h : ¬A → A) : A := by tauto`
This theorem shows that `tauto` works on substitution instances of a tautology.
`theorem T0b {a : Nat} : ¬a = 0 ∨ a = 0 := by tauto`

8 Induction

8.1 Sum of a sequence

In this section we will prove theorems using induction. The linear display of expressions in Infoview is hard to follow so we will write the expressions in mathematical notation.

The first program we prove is the one usually used to introduce induction.

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

We import `Finset` for finite sets and `BigOperators` which enables the use of the operators ∑ (`\sum`) and ∏ (`\prod`).

```
import Mathlib.Tactic
import Mathlib.Data.Nat.Basic
import Mathlib.Algebra.BigOperators.Basic
namespace gentle
open Nat Finset BigOperators
```

Tip: `Finset`

Given a finite set of numbers `s` and a function `f` over these numbers, `Finset` defines the expressions `∑ x in s f x` and `∏ x in s f x`.

Tip: `range`

`range n` is defined as the set of natural numbers less than `n`.

Since `range n` is $1,2,\ldots,n-1$, to get the set of natural numbers $1,2,\ldots,n$ the proof will use `range (n + 1)`. We will use $r(n+1)$ for this sequence of numbers:

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

```
theorem sum_id (n : ℕ) :
    ∑ i in range (n + 1), i = n * (n + 1) / 2 := by
  symm
  have : 0 < 2 := by norm_num
```

This first steps in the proof are to replace the goal with the symmetric goal

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

and to prove the trivial hypothesis $0 < 2$. The hypothesis is *anonymous* and is subsequently accessed as `this`.

Tactic: `symm`

Replace an equality `t = u` by `u = t`. The tactic works for any symmetric relation.

Tip: Anonymous hypothesis

A hypothesis can be unamed and can subsequently be accessed as `this`.

Next, division is replaced by multiplication which is easier to work with.

$$n(n+1) = 2 \sum_{i=r(n+1)} i\,.$$

```
  apply Nat.div_eq_of_eq_mul_right this
    -- div_eq_of_eq_mul_right:
    --    (0 < n) ∧ (m = n * k) → m / n = k
```

The proof continues by induction, where `k` is the variable of induction and `ih` is the inductive hypothesis.

```
  induction' n with k ih
```

Tactic: `induction'`

To prove a goal with `n` by induction, specify the induction variable `k` and a name `ih` for the inductive hypothesis.

The base case for $k = 0$ is proved using Lean's simplifier.

$$0(0+1) = 2 * \sum_{i=0(0+1)} i = 2 \cdot 0(0+1)$$

```
  · simp
```

Tactic: `simp`

The `simp` tactic uses lemmas and hypotheses in `Mathlib` to simplify a goal or hypothesis.

For the inductive step we assume that the theorem holds for k and prove that it holds for $k+1$. The proof starts by separating out the last term of the sum. The notation is a bit clumsy because we aren't replacing $(k+1)+1$ by $k+2$.

$$(k+1)((k+1)+1) = 2 \sum_{i=r((k+1)+1)} i = 2\left(\sum_{i=r(k+1)} i + (k+1)\right).$$

```
  · rw [Finset.sum_range_succ]
      -- Finset.sum_range_succ:
      -- ∑ x in range (n + 1), f x = ∑ x in range n, f x + f n
```

Next the distributive law is used.

$$(k+1)((k+1)+1) = 2 \sum_{i=r(n)} i + 2(k+1)\,.$$

```
    rw [mul_add 2]
      -- mul_add: a * (b + c) = a * b + a * c
```

Substitute the inductive hypothesis.

$$(k+1)((k+1)+1) = k*(k+1) + 2(k+1)\,.$$

```
    rw [← ih]
```

Lean is using the successor function which is now changed to $+1$ and the proof concludes with a simple calculation that can be performed by the tactic `ring`.

```
    rw [succ_eq_add_one]
      -- succ_eq_add_one : succ n = n + 1
    ring
  done
```

8.2 Inductive definition of addition

This subsection demonstrates inductive definitions and the tactic `rfl` (*reflexive*).

Here is an inductive definition of natural numbers. The program is enclosed in the namespace `gentle` to avoid clashing with predefined `Nat`. After defining `Nat` we open its namespace so that we can use `zero` and `succ` without prefixes.

```
namespace gentle

inductive Nat
| zero : Nat
| succ : Nat → Nat

namespace Nat
```

The next step is to defined addition inductively.

```
def add : Nat → Nat → Nat
  | x, zero => x
  | x, succ y => succ (add x y)
```

Although we normally think of addition as a function of two variables $add(a,b)$, there are advantages to defining it as a function that takes one variable $add(a)$ resulting in another function $add_a(b)$ that gives the sum of a and b.[3]

The first theorem we proof is $0+n=n$ for all natural numbers n. The proof is by induction and the base case $0+0=0$ follows by `rfl` from the base case of the definition of `add`.

[3]This is called *currying* but an explanation is outside the scope of this tutorial.

```
theorem zero_add (n : Nat) :
    add zero n = n := by
  induction' n with k ih
  · rfl
```

Tactic: `rfl`

`rfl` solves a goal that is an equality if they are *definitionally equivalent*, that is, they define the same entity.

Consider the goal that appears below:

`((m + k) + 1) + 1 = (m + (k + 1)) + 1`

By definition of `add` above, `(a + (b + 1)` is `(a + b) + 1` so the right-hand side of the equation `(m + (k + 1)) + 1` means `((m + k) + 1) + 1` which is the left-hand side. Therefore, the tactic `rfl` solves it. You should similarly check the other uses of `rfl` below.

The inductive step is to prove $0 + (k + 1) = (0 + k) + 1$. Using definition of `add` to rewrite the left-hand side, we get $(0 + k) + 1 = k + 1$ which can be solved using the inductive hypothesis that $0 + k = k$.

```
  · rw [add]
    rw [ih]
  done
```

Next we prove $(m + 1) + n = (m + n) + 1$ by induction. The base case $(m + 1) + 0 = m + 1$ is immediate from the base case of the definition of `add`.

```
theorem succ_add (m n : Nat) :
    add (succ m) n = succ (add m n) := by
  induction' n with k ih
  · rfl
```

The inductive step is to prove $(m + 1) + (k + 1) = (m + (k + 1)) + 1$. From the definition of `add` it follows that $((m + 1) + k) + 1 = (m + (k + 1)) + 1$.

```
  · rw [add]
```

Substituting the inductive hypothesis $(m + 1) + k = (m + k) + 1$ results in $((m + k) + 1) + 1 = (m + (k + 1)) + 1$, which, as noted above, is definitionally equivalent and can be solved by `rfl`.

```
    rw [ih]
    rfl
  done
```

8.3 Multiple steps in a proof

In this section we explain two ways to shorten proofs in Lean by performing more than one step in the same action. We start with the proof of $(m|n \vee m|k) \rightarrow m|mk$ using methods already encountered.

Since the hypothesis is a disjunction, the goal must be proved separately for each disjunct using `rcases` to create separate hypotheses. Furthermore, since $m|n$ is defined as $\exists a(n = ma)$ so new variables must be introduced (page 16).

```
theorem div_prod1 {m n k : ℕ} (h : m ∣ n ∨ m ∣ k) :
    m ∣ n * k := by
  rcases h with ⟨a, h1⟩ | ⟨b, h2⟩
```

The new hypotheses are `h1 : n = m * a` and `h2 : k = m * b` which can be used to rewrite the goals. The rest of the proof is straightforward.

```
  · rw [h1]
    rw [mul_assoc]
    apply dvd_mul_right
      -- dvd_mul_right: a ∣ a * b
  · rw [h2]
    rw [mul_comm]
    rw [mul_assoc]
    apply dvd_mul_right
  done
```

Proofs can be shortened by sequentially rewrites of the goal.

```
· rw [h1, mul_assoc]
· rw [h2, mul_comm, mul_assoc]
```

The hypotheses `h1` and `h2` replace `m ∣ n` and `m ∣ k` by the definitionally equivalent `n = m * a` and `k = m * b`, so `rfl` can be used to perform the substitutions without creating hypotheses.

```
theorem div_prod2 {m n k : ℕ} (h : m ∣ n ∨ m ∣ k) :
    m ∣ n * k := by
  rcases h with ⟨a, rfl⟩ | ⟨b, rfl⟩
  · rw [mul_assoc]
    apply dvd_mul_right
  · rw [mul_comm, mul_assoc]
    apply dvd_mul_right
  done
```

A Keyboard shortcuts

Lean uses Unicode symbols that are not on keyboards. This table shows keyboard shortcuts that are used in VSC for entering the symbols. Enter a space or tab after a shortcut and the symbol will appear. If you hover over an symbol the shortcut will be displayed.

Unicode symbol	Keyboard shortcut
∣	`\\|`
ℕ	`\N`
≤	`\le`, `\leq`
≥	`\ge`, `\geq`
≠	`\ne`, `\neq`
←	`\l`
→	`\r`, `\imp`
↔	`\iff`
∧	`\and`
∨	`\or`
∀	`\all`, `\forall`
∃	`\ex`, `\exists`
¬	`\n`, `\neq`
⟨	`\<`
⟩	`\>`
⊢	`\\|-`, `\vdash`
h_1	`h\1`
∑	`\sum`
∏	`\prod`

B Tactics

Tactic	Page	Definition
`apply`	4	If you have a theorem `P→Q` and the goal matches `Q` then `apply ⊢P→Q` removes the goal `Q` and adds `P` as the new goal. If you have a theorem `Q` and the goal matches `Q` then `apply Q` removes the goal and there are no more goals.
`by_contra`	22	This tactic removes a goal `P`, adds the hypothesis `¬P` and creates a new goal `False`.
`contrapose`	23	Transforms a goal into its contrapositive. Applied to a hypothesis, it makes the negation of the goal into a hypothesis and the negation of the hypothesis into the goal. An exclamation point following `contrapose` calls tactic `push\_neg` on the resulting contrapositive.
`constructor`	19	Splits a goal into two subgoals: equivalence (↔) into two implications (Modus ponens and Modus ponens reversed) and conjunction (∧) into two conjuncts.
`exact`	9	The tactic `exact h` is used when `h` is exactly the statement of the current goal, so the goal is now proved. `exact` is similar to `apply` but limited because it can only be used if the hypothesis exactly matches the goal.
`have`	6	Introduces a new named hypothesis for use in the proof.
`induction'`	26	To prove a goal with `n` by induction, specify the induction variable `k` and a name `ih` for the inductive hypothesis.
`intro`	6	Introduces free variables in place of bound variables in a universally quantified formula. A universally bound variable means that the formula has to hold for an arbitrary value so we simply substitute a variable that is this arbitrary value. If the goal is `⊢P→Q` then `intro` introduces `P` as a hypothesis. If `Q` is proven then the hypothesis can be *discharged* and `P→Q` is proven. For an existential goal `∃ a A(a)`, `intro c` introduces `A(c)` as the new goal.
`left, right`	21	If the goal is a disjunction, tell Lean which disjunction you want to prove.
`linarith`	21	The tactic solves linear equalities and inequalities. Unlike `ring` it can use hypotheses and unlike `norm_num` it can solve equations with variables.
`norm_num`	18	Proves numerical equalities and inequalities that do not use variables.
`range`	25	`range n` is defined as the set of natural numbers less than *n*.
`rcases`	16	Given a hypothesis or goal that is a disjunction `A∨B` the tactic `rcases` splits it into two sub-hypotheses or subgoals `A` and `B`. For a disjunctive hypothesis, prove the goal under both sub-hypotheses. For a disjunctive goal, prove one of the subgoals.
`rcases`	17	`rcases h with ⟨v, h'⟩` means given a hypothesis `h`, let `v` be a value such that the new goal is `h'`.
`rfl`	28	`rfl` solves a goal that is an equality if they are *definitionally equivalent*, that is, they define the same entity.

`ring`	17	Proves equalities that can be proved directly from the axioms of a commutative ring without taking any hypotheses into account. For the ring of integers, addition, subtraction, multiplication and powers by natural numbers, but not division, can be used.
`rintro`	20	Performs `intro` and then `rcases` to split the resulting hypothesis.
`rw`	8	`rw [eqn]`, where `eqn` is an equation or an equivalence, rewrites the goal by replacing occurrences of the the left-hand side of `eqn` with the right-hand side. `rw [← eqn]` rewrites the goal by replacing occurrences of the right-hand side of `eqn` with the left-hand side. `rw [eqn] h` and `rw [← eqn] h` rewrite expressions in the hypothesis `h`.
`simp`	26	The `simp` tactic uses lemmas and hypotheses in `Mathlib` to simplify a goal or hypothesis.
`sorry`	14	Proves any theorem.
`symm`	25	Replace an equality `t = u` by `u = t`. The tactic works for any symmetric relation.
`use`	14	Given an existential goal `∃ c A(c)`, this tactic introduces a free variable for the bound variable `c` to form a new goal. It will also attempt to prove the goal using the hypotheses.

C Tips

Topic	Page	Tip
Anonymous hypothesis	26	A hypothesis can be unamed and can be subsequently accessed as `this`.
Associativity	8	The default associativity in Lean is *left*: `a + b + c` means `(a + b) + c`. Even if the operation is associative, you still have to prove associativity for a given expression.
Division	5	The division operator in Lean is not the `|` symbol on your keyboard, but a similar Unicode symbol `|` obtained by typing `\\|`.
`done`	5	All proofs should be terminated by `done`. This is not necessary but if your proof is not complete, `done` will display a message.
Don't care	23	When a name or value is syntactically required but you don't care what its value is, you can use the underscore symbol `(\_)` instead.
`Finset`	25	Given a finite set of numbers `s` and a function `f` over these numbers, `Finset` defines the expressions `∑ x in s f x` and `∏ x in s f x`.
Indentation	6	All statements used to prove a hypothesis must be indented a fixed number of spaces.
Infoview	1	Enter `ctrl-shift-enter` to open the Lean Infoview where hypotheses, goals and errors are displayed. I have found it helpful to place the cursor just before the source line, and then to alternate between `Home` and `End` while looking at the Infoview.
Implicit	9	Parameters declared with braces, such as `{a b c : Int}` state that `a b c` are *implicit* bound variables in the theorem whose actual names and types can be inferred when the theorem is used.
Negation	12	A negation `¬p` is defined as `p → False` so it is proved by introducing `p` as a hypothesis and then showing that this results in a contradiction by proving `False`.
`range`	25	`range n` is defined as the set of natural numbers less than `n`.
Right-to-left	12	The symbol `<|` means to compute the expression to its right and pass it to the left.
Tactic	1	If you hover over a tactic its specification will be displayed.
Theorem	2	You can display the statement of a theorem by hovering over its name.