Masterarbeit

Freie Universität Berlin

Formal proofs from THE BOOK

Some proofs from "Proofs from THE BOOK", by the book

Yves Jäckle

December 5th, 2023

Supervision:
Prof. Dr. Sebastian Pokutta & Prof. Dr. Tibor Szabó

Abstract

This thesis accompanies code formalising content from the textbook "Proofs from THE BOOK" by M. Aigner and G. Ziegler in the interactive theorem proving language Lean. We first introduce the interactive theorem prover (ITP) Lean and the textbook "Proofs from THE BOOK". Then, we illustrate how one interacts with Lean and exemplify the practice of formally verifying theorems with it. This is followed by a short discussion of of the logical and mathematical foundations of Lean. To conclude the introduction, we discuss the notion of *tactics* and that of metaprogramming in Lean. Next, a section is dedicated to a detailed discussion of the formalisation process: we discuss the practice of theorem proving in Lean, its peculiarities and challenges. We illustrate this discussion in a new section containing three case studies of formalisations of content from "Proofs from THE BOOK". The mathematical areas covered by these studies are number theory, incidence geometry and extremal graph theory. Finally, to conclude the thesis, we point to ongoing projects surrounding the development of Lean and of formalised mathematics, and in particular to research efforts aiming to automate the process of formal theorem proving.

Contents

1 Authors notes

1.1 The code

The main object of this thesis is code written in Lean. This code is publicly available on the GitHub repository we link to in reference [4]. We recommend installing Lean 3 (refer to instructions on [1]) or using the online editor available at [2], so as to interact with the code written for this thesis. Indeed, Lean is a so-called *interactive theorem prover* (ITP), which allows the user to follow what each line of code of a formal proof does. Note however, that we have tried to design this thesis so that it may be read without referring to the code, or interacting with Lean. Installation and usage instructions are available in the material we link to.

To use the online editor, we recommend copy-pasting the entirety of the content of a given file of our repository that we refer to. The editor may require a while to load, and we have experienced some technical problems with it. In the case of an error message, it may help to cut the command lines starting with `import`, and pasting them back into the code at the same spot, to get the editor to function properly.
Alternatively, the reader may install Lean on a personal device. After installation, interacting with Lean is possible through the use of *Visual Studio Code* (VS Code ; refer to [3]), and the Lean extension for it.

Throughout the thesis, we will append references made in a section at the end of that section.

References:

1 https://leanprover-community.github.io/lean3/

2 https://leanprover-community.github.io/lean-web-editor/

3 https://code.visualstudio.com/

4 https://github.com/Happyves/Master_Thesis

1.2 Thanks

I would like to thank the following people:

- Christoph Spiegel for introducing me to Lean and advising and supporting me in all sorts of matters, in particular the IT aspects and coding in general, both of which I was not familiar with.
- Yaël Dillies for answering my beginners questions and offering advice and directions.
  My early questions must have been boring and tiresome to answer, and yet, they took the time to answer them. Also, I made use of `yael_search`, quite often.
- The Xena project discord server members, in particular Kevin Buzzard, Bhavik Mehta, Eric Wieser, David Ang, and the users by the names "ericrbg" and "metahumor". The Xena project discord server is a wonderful community that provided me with solutions and directions on many occasions.
- The Lean and mathlib communities. Formalising math is a community effort, and each step of its process, each contribution, is essential. There is no other science I know of that serves as a better example for this statement.
- My advisors Prof. Sebastian Pokutta and Prof. Tibor Szabó for allowing me to work on this project.

Thank you!

2 Introduction

2.1 Lean and ITPs

Lean
For our purposes, Lean [1] is an interactive formal theorem proving language (abbreviated as ITP), initiated by Leonardo de Moura in 2014. We remark that Lean may also be used as a functional programming language.
For a retrospective on the development of Lean, we refer to section *"1.3.4 A Short History of Lean"* of [11].

Interactive theorem proving
ITPs allows users to state and prove mathematical statements in a language that is then "compiled" (type-checked), to verify its correctness. There are many ITPs available besides Lean: two prominent ones are Isabelle [5] and Coq [6]. The earliest ITPs, such as SAM and Automath, were developed in the 1960s. ITPs differ in their foundational logic, their language and the way it is compiled, as well as their community's size, interests and policies. Lean, for instance, is founded on the so-called *Calculus of Inductive Constructions (CiC)*, a certain *type theory*. We shall take note of two aspects of Lean's community: it is focused on classical mathematics (as opposed to constructive mathematics [12]), and attempts to maintain a centralised and coherent library named *mathlib*.
We refer to the surveys [3] and [4] listed below for a more detailed exposition of ITPs and their history.
Interest in ITPs is generated by two, among many more, reasons we now elaborate.

Formal verification is the highest proof standard a mathematical proof can have. Mistakes in mathematical publication can make it past the peer-review process. The expository paper [4] provides some examples for this problem. The mechanical nature of formal verification, on the other hand, ensures an uncircumventable degree of rigour in proofs that prohibits errors. When writing the code for this thesis, we made - more often then we would have liked to - minor mistakes in the statements of lemmata and theorems. One usually becomes aware of these mistakes when trying to perform a certain step of a proof and by then realizing, that a necessary condition for this step was missing in the statements hypotheses. These mistakes are quite common, and are almost invisible in informal proofs: in his blog-post on formalising a 2023 result in Ramsey theory [14], Bhavik Mehta suggests calling such errors "mathematical typos". We found that interactive formal verification serves as a gentle automatic peer-review: the interaction allows for possible errors to be easily recognised, step by step.

Projects such as the *Liquid Tensor Experiment* [7] initiated by Peter Scholze have shown that relevant and recent research can be formalised in Lean. This has raised an interest in the mathematical community for formal verification at a mainstream level. It allows envisioning a future in which formal verification is integrated to the peer-review process, thereby minimising the risk of errors in journal publications.

We now present the second reason of interest. The nature of ITPs allows for proof-writing to be automated, up to various degrees. A first form of automation can be found in Lean's so-called *tactics*, which are programs that write short proofs for statements that allow for an algorithmic proof procedure, such as algebraic manipulations. With the increase in publicly available formalised mathematics, researchers of the various fields of AI are gaining interest in using this data to train different models at generating proofs for prescribed statements. An example of such a model is *LeanDojo*, which was developed by a group of researchers represented by Kaiyu Yang, whose preprint, published in June 2023 and available at [8], is still under review. We refer to the survey [9] for a presentation of a particular branch of AI related to ITPs.

Mathlib
Finally, to close this introduction to ITPs, we mention an important aspect to formal mathematics: the community. Lean's mathematical foundations and a collection of fundamental definitions, lemmata and theorems are publicly available on the library named *mathlib* [10]. This is a collection of code written by various authors, that may be used to prove theorems of ones own. In this Master thesis, we make use of mathlib and refer to it on many occasions. Contributing to developing mathlib by engaging with the *Lean community* [2] is possible and encouraged. The community communicates via a social network named *Zulip chat*, which can be found on [13].

References:

1 https://lean-lang.org/

2 https://leanprover-community.github.io/lean3/

3 Maric, Filip. "A survey of interactive theorem proving." Zbornik radova 18.26 (2015): 173-223.

4 Avigad, Jeremy, and John Harrison. "Formally verified mathematics." Communications of the ACM 57.4 (2014): 66-75.

5 https://isabelle.in.tum.de/

6 https://coq.inria.fr/

7 https://leanprover-community.github.io/liquid/

8 https://leandojo.org/

9 Kühlwein, D., and J. C. Blanchette. "A Survey of Axiom Selection as a Machine Learning Problem." (2014).

10 https://github.com/leanprover-community/mathlib

11 Ullrich, Sebastian Andreas. An Extensible Theorem Proving Frontend. Diss. Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2023

12 Bauer, Andrej. "Five stages of accepting constructive mathematics." Bulletin of the American Mathematical Society 54.3 (2017): 481-498

13 https://leanprover.zulipchat.com/

14 https://xenaproject.wordpress.com/2023/11/04/formalising-modern-research-mathematics-in-real-time/

2.2 Proofs from "Proof from THE BOOK", by the book

"Proofs from THE BOOK" (Fifth Edition), is a book written by Martin Aigner and Günter Ziegler and published by Springer in 2014 (the first edition was published in 1999). It's a collection of beautiful proofs from a wide range of areas of math. The book's title refers to an expression used by mathematician Paul Erdös (P.G.O.M.L.D.A.D.L.D.C.D (refer to the english Wikipedia article on Erdös to find out what this means)). "THE BOOK" is a book owned by god, which contains all the most beautiful proofs of mathematics, and which is kept hidden from humanity.
Faced with a beautiful proof, Erdös would exclaim "This one's from THE BOOK!".
We will see that in god's book of proofs, the devil is in the details.

The purpose of this thesis is to formalise some of the proofs from "Proofs from THE BOOK" in Lean.
We want to formally verify the proofs and gain experience with formalisation in the process. Our initial goal was to gain an understanding for the differences between formal and informal proofs. We wanted to judge the capacities of Lean, and the determine potential challenges and misconceptions of formalising mathematics in Lean. Pleased with our findings, our goal was moved to that of acquiring practical experience in the latter task. Our work is also intended to be used for pedagogical purposes: its content makes for good introductory examples. We make the remark that Lean is a relatively young language, and that there is relatively little introductory material available for it: thus, there is a demand for content such as the one we develop.

The content we formalise can be found in chapters 1, 11 and 27 for "Proofs from THE BOOK" (Fifth Edition). In our repository, the files whose names start with "`FormalBook`" correspond to this formalised content. In each file, we include a preamble describing the content of the file and containing a list of the mathematical statements to be found in this file. These statements are listed according to the names we gave them, and each entry of the list contains a short informal description of the corresponding statement.
Our choice of which content to formalised was guided by two criteria: simplicity and multi-disciplinarity. Indeed, we chose content that is sufficiently elementary, in the sense that it does not depend on advanced results, so as to have full mathlib support. In "Proofs from THE BOOK", chapter 1 covers number theory, chapter 11 covers geometry and chapter 27 covers combinatorics and graph theory. Working on these topics allowed us to explore multiple areas of mathlib, thereby gaining a broad culture not bound to a specific field.

We shall now list the files formalising content from "Proofs from THE BOOK", with short descriptions and references to the corresponding location of the content in "Proofs from THE BOOK".

- `FormalBook_Ch1_InfinitudeOfPrimes_1stProof`
  Formalises the proof entitled "*Euclid's proof*" in Chapter 1 "Six proofs of the infinity of primes".
- `FormalBook_Ch1_InfinitudeOfPrimes_2ndProof`
  Formalises the proof entitled "*Second proof*" in Chapter 1 "Six proofs of the infinity of primes".
- `FormalBook_Ch1_InfinitudeOfPrimes_3rdProof`
  Formalises the proof entitled "*Third proof*" in Chapter 1 "Six proofs of the infinity of primes".
- `FormalBook_Ch1_InfinitudeOfPrimes_4thProof`
  Formalises the proof entitled "*Fourth proof*" in Chapter 1 "Six proofs of the infinity of primes".
- `FormalBook_Ch1_InfinitudeOfPrimes_5thProof`
  Formalises the proof entitled "*Fifth proof*" in Chapter 1 "Six proofs of the infinity of primes".
- `FormalBook_Ch11_LinesInThePlane_SylvesterGallai`
  Formalises the proof of the theorem entitled "*Theorem 1*" in Chapter 11 "Lines in the plane and decompositions of graphs".
- `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry`
  Formalises the proofs of the theorems entitled "*Theorem 2*" and "*Theorem 3*", as well as that to the claim on page 75, written in italic, in Chapter 11 "Lines in the plane and decompositions of graphs".
- `FormalBook_Ch27_PigeonholeDoublecounting_numbers`
  Formalises the proofs of the claims of the section entitled "*1. Numbers*" in Chapter 27 "Pigeon-hole and double counting".
- `FormalBook_Ch27_PigeonholeDoublecounting_sequences`
  Formalises the proof of the claim of the section entitled "*2. Sequences*" in Chapter 27 "Pigeon-hole and double counting".
- `FormalBook_Ch27_PigeonholeDoublecounting_sums`
  Formalises the proof of the claim of the section entitled "*3. Sums*" in Chapter 27 "Pigeon-hole and double counting".
- `FormalBook_Ch27_PigeonholeDoublecounting_numbers_again`
  Formalises the proof of the claim of the section entitled "*4. Numbers again*" in Chapter 27 "Pigeon-hole and double counting".
- `FormalBook_Ch27_PigeonholeDoublecounting_graphs`
  Formalises the proof of the theorem of the section entitled "*5. Graphs*" in Chapter 27 "Pigeon-hole and double counting".

2.3 Structure of the thesis

We have decided to give an exposition aimed at readers unfamiliar with Lean.
Our aim will therefore *not* be a discussion the details of our implementations, that may be beneficial to readers already familiar with Lean and mathlib, and in search of examples to illustrate the practice of formalisation. However, we include a document named "chapters.pdf" in our repository, in which each file of code is discussed at a level we deem more adequate for readers already familiar with Lean. In each of its chapters, we recall the informal statement and proof from the corresponding chapter of "Proofs from THE BOOK" to be formalised, and discuss our formalisation. We take note of helpful mathlib lemmata that we visually highlight through the use of blue boxes.

Our aim for this thesis will be to introduce the reader to Lean and to formalising mathematics with it. We have also made the decision to keep this introduction somewhat superficial: this thesis does not aim at teaching Lean. Readers desiring to learn the language may find better and lengthier treatments in [1] and [2].
As a consequence, we use only parts of our code to illustrate aspects of formalisation we deem interesting.

We propose the following structure to achieve our goals.
First, in section 3, titled *Lean by examples*, we introduce the reader to Lean. We do so with a first example of a proof, whose purpose is to illustrate how one uses and interacts with Lean. Then, we give a second, more sophisticated example of a proof, which is based on content from "Proofs from THE BOOK". In this example, we give a highly detailed, line-by-line, description of the code, and compare it to its informal source. Moving on, we give examples of foundations in Lean: we discuss the formal foundations of natural numbers, as well as operations and relations on them. We also address an important concept of type theory usually referred to as the *Curry-Howard correspondence*, or *propositions-as-types principle*. To close the section, we discuss *tactics* and *metaprogramming*.

In section 4, titled *The formalisation process*, we discuss said process, its challenges and some potential misconceptions surrounding it. Many of the topics treated in this section cannot be read form the code we produced: they occurred during the production of the code, and are invisible in the final product. Seeing as they are not bound to specific cases, but reflect the global experience of formalisation, we have not integrated them to the section that follows.

In section 5, titled *Case studies*, we illustrate the formalisation process on 3 examples of formalisations of content from "Proofs from THE BOOK". We have chosen, for these examples, the most challenging ones among all our produced implementations. To be precise, we chose examples that best illustrate the challenges of formalisation. We wish to make the immediate remark that this may portray formalisation as a difficult task. This is not our intent: we merely selected these examples as we believe them to be the most interesting to discuss, not because we believe them to be representative of the typical formalisation experience. For an example of a more common formalisation experience, we refer to the content of our section 3.2 titled *Example of a (longer) proof*.
Section 5 will thus not contain a detailed discussion of the code, but a discussion of the different decisions we made in our formalisation process, and a discussion of the obstacles we faced when trying to implement our decisions.

Finally, we draw a short conclusion whose aim is to discuss the development of Lean. We present current projects surrounding Lean and address opportunities for contributing to the Lean community.

References:

1 Theorem Proving in Lean 4:
https://lean-lang.org/theorem_proving_in_lean4/

2 Mathematics in Lean:
https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf

3 Lean by examples

We encourage the reader to consult the file named `Introduction` in the `src` folder of our repository. Readers that have installed Lean may clone/download the repository and open the file in VS Code. Readers using the online editor may copy-paste the entire content of the file and replace the editor's default code with it. We recall that it is also possible to read what follows without referring to the code.

3.1 Example of a (short) proof

First, we will give an example of the statement and proof of a lemma in Lean. We will step through the code, explaining what the code does and how this translates to informal math. We have included screenshots of VS Code, so as to teach/describe to the reader how to interact with Lean.

We shall prove that the sum of two odd natural numbers is even. This is stated as follows:

```
lemma my_lemma
  (x y : ℕ) (hx : odd x) (hy : odd y):
  even (x+y) :=
```

Here, the keyword `lemma` declares that we are about to state and prove a lemma, and `my_lemma` is the name we will refer it with, if we ever need it in further code. Next, we declare the objects and assumptions on these objects that the lemma makes use of. We declare two natural numbers we name x and y, with `(x y : ℕ)`. We then assume for each that it is odd: for x, for examples, we declare `(hx : odd x)`, so as to have a hypothesis we named hx, which certifies that x is odd. The definition of odd and even numbers is provided by mathlib, the publicly available code library collecting many elementary definitions and lemmata.
On the line that follows, we write the statement that is the result of the lemma, and which we set out to prove. In our example, it is `even (x+y)`, which states that the sum of x and y is an even number.

Now, we will write a proof for this lemma. The proof is contained between the keywords `begin` and `end`.

```
begin
  -- Click here and look at the infoview
  rw nat.even_iff,
  rw nat.odd_iff at hx hy,
  rw nat.add_mod,
  rw [hx, hy],
  norm_num,
end
```

The first line, which starts with two minuses, is a comment: this line will not be read as code, it is intended only for the reader, not the machine. In VS Code, the situation will be the following:

The panel on the right is the so-called *infoview*. It will display the current *proof state*. The names represented in yellow (gold ?) are the objects and hypotheses of the lemma, and the statement preceded by the symbol ⊢, is the goal, the result to be proven.

We shall now make the first step in our proof, in line `rw nat.even_iff`. Here, `nat.even_iff` the the name of a lemma from mathlib, which states `∀ {n : ℕ}, even n ↔ n % 2 = 0`. The symbol `%` in `a%b` denotes the remainder in the Euclidean division of a by b. So this mathlib lemma states that a number is even if and only if its remainder by 2 is 0. The command `rw`, which stands for `rewrite`, uses this lemma to change the goal to its equivalent statement, according to the equivalence that is `nat.even_iff`.
Consulting the infoview after clicking after this line of code yields:

We note that the goal has changed to an equivalent form.

On the next line, we use mathlib lemma `nat.odd_iff : ∀ {n : ℕ}, odd n ↔ n % 2 = 1`, to rewrite the hypotheses hx and `hy`, so that they then state $x\%2 = 1$ and $y\%2 = 1$ respectively. Note the use of the `at` keyword to specify which hypotheses we want the rewrite to occur at.

On the line that follows, we use `nat.add_mod : ∀ (a b n : ℕ), (a + b) % n = (a % n + b % n) % n`. The goal now states $(x\%2 + y\%2)\%n = 0$. With the line preceding the final one, we let Lean find the expressions $x\%2$ and $y\%2$ in the expression of our goal, and have them replaced by the equal expressions provided by equalities `hx` and `hy`.

At this stage, the goal states $(1 + 1)\%2 = 0$. Showing this is merely computation (for our expository purposes), so we may use the so-called *tactic* named `norm_num` to certify that this computation is correct. The infoview is now:

This means that the proof is complete!

Conclusion:
This section served to orient the reader as to what Lean is, and as to how it is used. Key concept to learn from this section are the notions of the infoview, the proof states with its goal and its hypotheses. In the next section, we give a example of a more involved proof, to exemplify what a larger formalisation project may look like.

3.2 Example of a (longer) proof

We discuss code contained in the file `FormalBook_Ch27_PigeonholeDoublecounting_numbers` of our repository. Our aim is to formalise the following theorem and its proof:

Appearance of coprimes

No matter how we pick $n+1$ numbers among the numbers $1, 2, ..., 2n$,
there will always be two picked number that are coprime.

Proof: Seeing as a number and its successor are coprime, it is enough to prove that by picking $n+1$ numbers among the numbers $1, 2, ..., 2n$, we always find a number and its successor.
To do so, we consider $1, 2, ..., 2n$ to be pigeons, and pairs $\{1, 2\}, ...\{2n-1, 2n\}$ to be pigeonholes.
There are n pigeonholes and $n+1$ pigeons, so that by the pigeonhole principle, one of the pairs $\{2k-1, 2k\}$ (for $k \in [n]$) must contain two pigeons. These are the desired number and its successor that we seek. □

Throughout the section, we will reproduce the code that implements this theorem and proof in Lean, piece by piece. In this code, the text following the double minuses, or between the backslashes, are comments that do not belong to the functional code. In these comments, we take note of the proof state, or changes to the proof state. Changes in hypotheses start with the name of the hypothesis, and changes to the goal start with ⊢.
Now, we shall discuss this code step by step.

In order to make use of many essential mathlib lemmata, we must import their corresponding files:

```
import data.finset.basic
import data.finset.card
import data.nat.gcd.basic
import data.nat.parity
import tactic
```

Now, we may start by formalising the theorem's statement. We name the theorem `claim_1`.

```
lemma claim_1
  (n : ℕ) (h : 1 ≤ n)
  (A : finset ℕ) (Adef : A ∈ (powerset_len (n+1) (Icc 1 (2*n)))) :
  ∃ a ∈ A, ∃ b ∈ A, (a≠b) ∧ (nat.coprime a b) :=
```

We declare a natural number `n`, and assume that $1 \leqslant n$, a fact we record under the name `h`. Indeed, mathlib's natural numbers start at 0, and we need to exlcude this case, as the statement is meaningless for it.
Next, we declare a finite set `A` of natural numbers, with command `(A : finset ℕ)`. This set will contain the $n+1$ numbers we pick among $1, 2, ..., 2n$. We express the latter fact through `(Adef : A ∈ (powerset_len (n+1) (Icc 1 (2*n))))`. Here, we produce an assumption we name `Adef`. We dissect its statement to understand its meaning. First, `Icc 1 (2*n)` denotes the finite set of all natural numbers x satisfying $1 \leqslant x \leqslant 2n$. Next, `(powerset_len (n+1) (Icc 1 (2*n)))` denotes the set of subsets of `Icc 1 (2*n)`, whose size is $n+1$. The fact that `A` is part of the latter set expresses the fact that it contains $n+1$ numbers chosen among numbers $1, 2, ..., 2n$.

Finally, following the `:`, we may state the conclusion of the theorem. With `∃ a ∈ A, ∃ b ∈ A,` we claim the existence of two numbers a and b that are members of `A`, and `(a≠b) ∧ (nat.coprime a b)` expresses their properties. The symbol ∧ represents the logical "and". With `(a≠b)` we claim the the numbers are distinct: the theorem is still correct without this condition, but it is weaker and does not reflect the informal statement.
The second condition `(nat.coprime a b)` describes the fact that the numbers are coprime. This definition is taken from `mathlib`, which will also provide us with many useful lemmata surrounding coprimality.

After writing `:=` we may start the proof in `begin-end` code block.

```
begin
  /-
  n: ℕ
  h: 1 ≤ n
  A: finset ℕ
  Adef: A ∈ powerset_len (n + 1) (Icc 1 (2 * n))
  ⊢ ∃ (a : ℕ) (H : a ∈ A) (b : ℕ) (H : b ∈ A), a ≠ b ∧ a.coprime b
  -/
  rw mem_powerset_len at Adef, -- Adef: A ⊆ Icc 1 (2 * n) ∧ A.card = n + 1
```

Our first step will be to "unfold", for lack of a better word, the definition of A. We do so with the mathlib lemma `mem_powerset_len`, so as to replace the statement of `Adef` with `A ⊆ Icc 1 (2 * n) ∧ A.card = n + 1`. Here, `A.card` denotes the size of `A`.

Before moving on with the formal proof, it is helpful to remind ourselves of the tools we intend to use, so as to plan our formalisation. The key lemma we will need is the pigeonhole principle which is available in mathlib under the name:

Pigeonhole principle
`finset.exists_ne_map_eq_of_card_lt_of_maps_to`

Given a map f mapping pigeons from a finite set s to a finite set t of pigeonholes, so that $|t| < |s|$,
we may conclude with the existence of two distinct pigeons x and y that are mapped to the same pigeonhole.
In code: `t.card < s.card → ∀ f : α → β, (∀ (a : α), a ∈ s → f a ∈ t) → (∃ (x : α) (H : x ∈ s) (y : α) (H : y ∈ s), x ≠ y ∧ f x = f y)`

The map we use to map pigeons to pigeonholes in our context will be $n \mapsto \lfloor \frac{n+1}{2} \rfloor$. The reader may convince themselves that 1 and 2 have image 1, and $2n-1$ and $2n$ have image n, for example.
In Lean, we may describe this map with `(λ (x : ℕ), (x+1) / 2)`. For natural numbers a and b, the number a/b is the quotient in the Euclidean division of a by b in Lean. Seeing as this quotient coincides with the floor of a fraction of natural numbers, we do indeed define $n \mapsto \lfloor \frac{n+1}{2} \rfloor$ with expression `(λ (x : ℕ), (x+1) / 2)`.

To proceed with our main formal proof, we shall apply first this principle with this map. We do so by defining a lemma internal to the proof with the keyword `have`, and we name it `Lem1`. This requires us to state the conclusion of the lemma, which we do as follows:

```
have Lem1 :
    ∃ a ∈ A, ∃ b ∈ A, (a≠b) ∧
    ((λ (x : ℕ), (x+1) / 2) a = (λ (x : ℕ), (x+1) / 2) b) :=
```

Note the similarity to the conclusion of the pigeonhole principle. The lemma will inherit the objects and assumptions from the context it is placed in. Its proof will be contained in the code-block delimited by `by { }`.

```
    by {let group_fn := (λ x, (x+1) / 2), -- group_fn: ℕ → ℕ := λ (x : ℕ), (x + 1) / 2
        have map_condition : (∀ a, a ∈ A → group_fn a ∈ (Icc 1 n)) :=
          by {-- ⊢ ∀ (a : ℕ), a ∈ A → group_fn a ∈ Icc 1 n
              intros x xdef,
              /-
              x: ℕ
              xdef: x ∈ A
              ⊢ group_fn x ∈ Icc 1 n
              -/
              dsimp [group_fn], -- (x + 1) / 2 ∈ Icc 1 n
```

```
            replace xdef := Adef.1 xdef, -- xdef: x ∈ Icc 1 (2 * n)
            rw mem_Icc at *,
            /-
            xdef: 1 ≤ x ∧ x ≤ 2 * n
            ⊢ 1 ≤ (x + 1) / 2 ∧ (x + 1) / 2 ≤ n
            -/
            split,
            {-- ⊢ 1 ≤ (x + 1) / 2
             rw nat.le_div_iff_mul_le,
             /-
             ⊢ 1 * 2 ≤ x + 1
             ⊢ 0 < 2
             -/
             linarith, -- goals accomplished
             norm_num, -- goals accomplished
             },
            {-- ⊢ (x + 1) / 2 ≤ n
             rw nat.div_le_iff_le_mul_add_pred,
             /-
             ⊢ x + 1 ≤ 2 * n + (2 - 1)
             ⊢ 0 < 2
             -/
             linarith, -- goals accomplished
             norm_num, -- goals accomplished
             },
           },
      -- map_condition: ∀ (a : ℕ), a ∈ A → group_fn a ∈ Icc 1 n
      apply exists_ne_map_eq_of_card_lt_of_maps_to _ map_condition,
      -- ⊢ (Icc 1 n).card < A.card
      rw nat.card_Icc, -- ⊢ n + 1 - 1 < A.card
      simp only [add_tsub_cancel_right], -- ⊢ n < A.card
      rw Adef.2, -- ⊢ n < n + 1
      simp only [lt_add_iff_pos_right, eq_self_iff_true, nat.lt_one_iff],
      -- goals accomplished
    },
/-
Lem1: ∃ (a : ℕ) (H : a ∈ A) (b : ℕ) (H : b ∈ A),
a ≠ b ∧ (λ (x : ℕ), (x + 1) / 2) a = (λ (x : ℕ), (x + 1) / 2) b
-/
```

We start the proof of this intermediate result with `let group_fn := (λ x, (x+1) / 2),`. This will define the function we will use for the pigeonhole principle, naming it `group_fn`. Then, we proceed with showing that this function maps numbers $1, 2, ..., 2n$ to numbers in $1, ..., n$. We do so in another intermediate lemma we name `map_condition`, which states `(∀ a, a ∈ A → group_fn a ∈ (Icc 1 n))`.
To prove this statement, we introduce a arbitrary natural number x, and the assumption that $x \in A$, which we name *xdef* as objects in our proof state, reducing the goal to `group_fn x ∈ (Icc 1 n))`, by writing `intros x xdef`. Next, we evaluate the function `group_fn` in x with line `dsimp [group_fn]`, so that the goal now states `(x + 1) / 2 ∈ Icc 1 n`. The line `replace xdef := Adef.1 xdef` will replace assumption `xdef`, which so far stated $x \in A$, with statement $x \in [2n]$. Next, we use line `rw mem_Icc at *,` to rewrite expressions of form `x ∈ Icc a b` into `a ≤ x ∧ x ≤ b`, at the current goal and at all current hypotheses. Assumption `xdef` now states `1 ≤ x ∧ x ≤ 2 * n`, and the goal now states `1 ≤ (x + 1) / 2 ∧ (x + 1) / 2 ≤ n`. With the line `split` that follows, we prove each part of the conjunction in the goal separately.

For first the goal, we use mathlib lemma `nat.le_div_iff_mul_le : 0 < k → (x ≤ y / k ↔ x * k ≤ y)`, so as to reduce goal `1 ≤ (x + 1) / 2` to goal `1 * 2 ≤ x + 1`, creating the additional goal $0 < 2$ in the process. What has happened is that Lean understood that in the lemma `0 < k → (x ≤ y / k ↔ x * k ≤ y)` we use to

rewrite our goal, $x := 1$, $y := x + 1$ and $k = 2$, and the condition $0 < k = 2$ must be verified.
The goal `1 * 2 ≤ x + 1` may be solved with the tactic `linarith`, which will recognize the fact that we have $1 \leqslant x$ among our assumptions and that we may derive the goal from it. Finally, the additional goal $0 < 2$ is handled by the `norm_num` tactic. We may now prove the second goal, `(x + 1) / 2 ≤ n`. We proceed quite similarly as in the previous step, this time using mathlib lemma `nat.div_le_iff_le_mul_add_pred : 0 < n → (m / n ≤ k ↔ m ≤ n * k + (n - 1))`, where Lean infers $n := 2$, $m := x + 1$ and $k := n$. This concludes the proof of the intermediate result `map_condition`, which certifies one of the conditions of the pigeonhole principle.

With `apply exists_ne_map_eq_of_card_lt_of_maps_to _ map_condition,` we declare that we shall use the pigeonhole principle to prove the goal of `Lem1`. Lean recognises which map f and which sets s and t to use in this context, with the help of the second of its conditions, `map_condition`. It now remains to prove the second condition of the pigeonhole principle in our context, which is `(Icc 1 n).card < A.card`.
We then use mathlib lemma `nat.card_Icc : ∀ (a b : ℕ), (Icc a b).card = b + 1 - a` to replace `(Icc 1 n).card` with $n + 1 - 1$ in our goal, which we `simp`lify to n through the use of `add_tsub_cancel_right : ∀ (a b : α), a + b - b = a`. In an almost final step, `rw Adef.2,` where `Adef.2` states `A.card = n+1`, we reduce the goal to $n < n + 1$. The last step consists of of some simplifications Lean provides us through the use of tactic `simp?`. This concludes our proof of `Lem1`.

At this stage, we know of the existence of two distinct number a and b in A which have the same value under $n \mapsto \lfloor \frac{n+1}{2} \rfloor$. We "unfold" this existence statement with line `rcases Lem1 with ⟨a, aA, b, bA, anb, abeq⟩`.

```
rcases Lem1 with ⟨a, aA, b, bA, anb, abeq⟩,
/-
a: ℕ
aA: a ∈ A
b: ℕ
bA: b ∈ A
anb: a ≠ b
abeq: (λ (x : ℕ), (x + 1) / 2) a = (λ (x : ℕ), (x + 1) / 2) b
-/
dsimp at abeq, -- abeq: (a + 1) / 2 = (b + 1) / 2
```

This has the effect of naming these numbers a and b, denoting the assumptions that they belong to A by `aA` and `bA` respectively, denoting the assumption that they are distinct by `anb` and denoting the assumption that that have the same value under the map from the pigeonhole principle by `abeq`. With command `dsimp at abeq,`, the latter assumption simplifies to `(a + 1) / 2 = (b + 1) / 2`.
Now, with the following lines, we make the claim that the a and b we just obtained are the two coprime numbers of A. We also remind Lean that a and b are members of A, and that they are distinct.

```
use a, split, use aA, -- ⊢ ∃ (b : ℕ) (H : b ∈ A), a ≠ b ∧ a.coprime b
use b, split, use bA, -- ⊢ a ≠ b ∧ a.coprime b
split, exact anb, -- ⊢ a.coprime b
```

Now, the goal is `a.coprime b`: we have to show that a and b are coprime. Recall that we will do so by showing that one is the successor of the other. Our strategy to achieve this latter goal will be to show that $a + 1$ and $b + 1$ must have different remainders in the division by 2. This implies the one of the remainders is 1 while the other is 0, so that the number whose remainder is 1 is the successor of the other, as they have the same quotient in the division by 2.

So, first, we show in intermediate lemma `Lem2` that $a + 1$ and $b + 1$ must have different remainders.

```
have Lem2 :
  (a+1)%2 ≠ (b+1)%2 :=
  by {-- ⊢ (a + 1) % 2 ≠ (b + 1) % 2
      intro con,
```

```
      /-
      con: (a + 1) % 2 = (b + 1) % 2
      ⊢ false
      -/
      have : a+1 = b+1 :=
        by {-- ⊢ a + 1 = b + 1
            rw ← nat.div_add_mod (a+1) 2, -- ⊢ 2 * ((a + 1) / 2) + (a + 1) % 2 = b + 1
            rw ← nat.div_add_mod (b+1) 2,
            -- ⊢ 2 * ((a + 1) / 2) + (a + 1) % 2 = 2 * ((b + 1) / 2) + (b + 1) % 2
            rw [abeq, con], -- goals accomplished
            },
      -- this: a + 1 = b + 1
      apply anb, -- ⊢ a = b
      exact nat.add_right_cancel this, -- goals accomplished
      },
  -- Lem2: (a + 1) % 2 ≠ (b + 1) % 2
```

We will show this by contradiction, through the use of line `intro con,`, which introduces assumption `con: (a + 1) % 2 = (b + 1) % 2` and sets the goal to `false`. With this assumption, we prove that $a+1 = b+1$, using mathlib lemma `nat.div_add_mod : ∀ (m k : ℕ), k * (m / k) + m % k = m` (which represents Euclidean division) and the fact that we have among our assumptions, the facts that the remainders and quotients of these two numbers by 2 are equal. Then, trough line `apply anb,` we declare that the contradiction will be derived by showing $a = b$, as we have the opposite among our assumptions. The final step of `Lem2`, which achieves this goal, is attained with mathlib's `nat.add_right_cancel : n + m = k + m → n = k`, where Lean infers $n := a$, $k := b$ and $m := 1$, and uses assumption `this` on the implication to derive the goal.

We now know that the remainders of $a+1$ and $b+1$ are different, but we do not know which is 1 and which is 0. To proceed, we use line `wlog H : (a+1)%2 < (b+1)%2 with Sym,`.

```
  wlog H : (a+1)%2 < (b+1)%2 with Sym,
  {/-
   Sym: ∀ (n : ℕ), 1 ≤ n → ∀ (A : finset ℕ), A ⊆ Icc 1 (2 * n) ∧ A.card = n + 1 →
        ∀ (a : ℕ), a ∈ A → ∀ (b : ℕ), b ∈ A → a ≠ b → (a + 1) / 2 = (b + 1) / 2 →
        (a + 1) % 2 ≠ (b + 1) % 2 → (a + 1) % 2 < (b + 1) % 2 → a.coprime b
   H: ¬(a + 1) % 2 < (b + 1) % 2
   ⊢ a.coprime b
   -/
   push_neg at H, --H: (b + 1) % 2 ≤ (a + 1) % 2
   rw ne_comm at *,
   /-
   anb: b ≠ a
   Lem2: (b + 1) % 2 ≠ (a + 1) % 2
   -/
   rw eq_comm at abeq, -- abeq: (b + 1) / 2 = (a + 1) / 2
   replace H := lt_of_le_of_ne H Lem2, -- H: (b + 1) % 2 < (a + 1) % 2
   specialize Sym n h A Adef,
   /-
   Sym: ∀ (a : ℕ), a ∈ A → ∀ (b : ℕ), b ∈ A → a ≠ b → (a + 1) / 2 = (b + 1) / 2 →
        (a + 1) % 2 ≠ (b + 1) % 2 → (a + 1) % 2 < (b + 1) % 2 → a.coprime b
   -/
   specialize Sym b bA a aA anb abeq Lem2 H, -- Sym: b.coprime a
   rw nat.coprime_comm, -- ⊢ b.coprime a
   exact Sym, -- goals accomplished
   },
  -- H : (a + 1) % 2 < (b + 1) % 2
```

What it does, is that it requires us to first show that if we have a proof of the fact that a and b are coprime, assuming `(a+1)%2 < (b+1)%2` (an assumption that will by named `H`), then we may also derive that a and b are coprime if the opposite of `(a+1)%2 < (b+1)%2` is true. Once this has been proven, we may proceed with the actual proof that a and b are coprime, with the additional assumption that `(a+1)%2 < (b+1)%2`. This encapsulates the mathematical "without loss of generality" (wlog ; hence the name).

So, we show that a and b are coprime if the opposite of `(a+1)%2 < (b+1)%2` is true, in the lines that follow. `push_neg at H,` has the effect of replacing the opposite of `(a+1)%2 < (b+1)%2` by `(b+1)%2 ⩽ (a+1)%2`. After some rewrites involving `ne_comm : a ≠ b ↔ b ≠ a` and `eq_comm : a = b ↔ b = a`, which will bring the concerned hypotheses in a from more suitable for future steps, we use mathlib's `lt_of_le_of_ne : a ≤ b → a ≠ b → a < b` to replace assumption `(b+1)%2 ⩽ (a+1)%2` by `(b+1)%2 < (a+1)%2`. Indeed, recall that by `Lem2`, we know that these remainders are distinct. In the two lines that follow, both starting with `specialize`, we make use of the assumption that we may show, for arbitrary x and y that x and y are comprime when assuming `(x+1)%2 < (y+1)%2`, to show that b and a are coprime: indeed, we set $x := b$ and $y := a$. Finally, with mathlib's `nat.coprime_comm : n.coprime m ↔ m.coprime n`, we may rapidly conclude that a and b are coprime in this context.

We are now back to showing that a and b are coprime, this time with the wlog assuption `(a+1)%2 < (b+1)%2`. The next step will consist in proving that `(a+1)%2 = 0` and `(b+1)%2 = 1`.

```
  have := nat.mod_lt (b+1) (show 0<2, by {norm_num,}), -- this: (b + 1) % 2 < 2
  interval_cases ((b+1)%2) with bcase,
  {-- bcase: (b + 1) % 2 = 0
   exfalso, -- ⊢ false
   rw bcase at H, -- H: (a + 1) % 2 < 0
   exact (nat.not_lt_zero _) H, -- goals accomplished
   },
```

To do so, we first set up assumption `(b + 1) % 2 < 2` with the help of mathlib lemma `nat.mod_lt :0 < y → x % y < y`. Then, we apply the tactic `interval_cases` to prove the lemma once for each of the values in $\{0, 1\}$ that `(b + 1) % 2` may have, as we just determined that `(b + 1) % 2 < 2`.
In the case where `(b + 1) % 2 = 0`, we will show that the assumptions are contradictory, so that the case cannot occur. This idea is implemented in the form of line `exfalso`. Rewriting the case's assumption at assumption `(a+1)%2 < (b+1)%2` will produce `(a + 1) % 2 < 0`, so that we may use mathlib's `nat.not_lt_zero : ∀ (a : ℕ), ¬a < 0` to derive the contradiction. Note that in the latter, `¬` represents logical negation.

We have arrived at the final code-block of the proof, which corresponds to case `(b + 1) % 2 = 1`.

```
  { -- bcase: (b + 1) % 2 = 1
   rw bcase at H, -- H: (a + 1) % 2 < 1
   rw nat.lt_one_iff at H, -- H: (a + 1) % 2 < 1
   rw nat.coprime_comm, -- ⊢ b.coprime a
   apply succ_coprime b a, -- ⊢ b = a + 1
   apply @nat.add_right_cancel _ 1 _, -- ⊢ b + 1 = a + 1 + 1
   rw ← nat.div_add_mod (a+1) 2, -- ⊢ b + 1 = 2 * ((a + 1) / 2) + (a + 1) % 2 + 1
   rw ← nat.div_add_mod (b+1) 2,
   -- ⊢ 2 * ((b + 1) / 2) + (b + 1) % 2 = 2 * ((a + 1) / 2) + (a + 1) % 2 + 1
   rw [abeq, bcase, H], -- ⊢ 2 * ((b + 1) / 2) + 1 = 2 * ((b + 1) / 2) + 0 + 1
   norm_num, -- goals accomplished
   },
  -- goals accomplished
end
```

After a rewrite of this case at `(a+1)%2 < (b+1)%2`, we use mathlib's `nat.lt_one_iff : n < 1 ↔ n = 0` to deduce that `(a + 1) % 2 = 0`. After a final use of `nat.coprime_comm` to order the numbers so as to use or lemmata correctly, we apply lemma `succ_coprime`, which states that a number and its successor are coprime. This will

reduce goal `b.coprime a` to goal `b = a + 1`. The lemma `succ_coprime` is not from mathlib: we prove it in the same file, before the proof of the theorem we discuss here.
After a use of `nat.add_right_cancel : n + m = k + m → n = k` to reduce that goal to `b+1=a+1+1`, we make use of `nat.div_add_mod` twice to reduce the goal further to `2 * ((b + 1) / 2) + (b + 1) % 2 = 2 * ((a + 1) / 2) + (a + 1) % 2 + 1`. Then, we replace the remainders with the values they have in our case and rewrite the quotients, which are equal by assumption, and conclude by a computation which verifies the equality, with `norm_num`.

Conclusion:
This example illustrates multiple aspects of formalisation. When comparing the size of the informal proof to that of the formal one, we note that the formal proof is much more spacious. The content of the proof justifies this fact: we had to show many facts the informal proof took for granted, and carefully justify some steps left to intuition in the informal proof. We also note that the syntax an structure of the formal proof in Lean is quite different from that of an informal proof. Our goal here was not to teach the reader Lean, its syntax or its commands, but rather to show that a certain technical knowledge is necessary to start writing proofs in Lean.

3.3 Example of foundations

We will now discuss the foundations of Lean superficially, so as to give the reader a taste of foundations, and reassuring them that they are solid. Indeed, any formal system for mathematics needs to define the most basic objects of mathematics, in a way that is both representative of its informal counterpart, and cannot be "hacked" to produce paradoxical or meaningless results. We will use natural numbers as a running example for this section.

Natural numbers are defined inductively, in a manner similar to the Peano axioms:

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

This means that numbers are "words" formed with the constant `zero` and the constructor (which is a function we may not evaluate) `succ`. For example, 2 is `nat.succ (nat.succ nat.zero)`. When we use the symbol 2 in Lean, the latter expression is what Lean understands. Logicians will call 2 *"syntactic sugar"* for the expression.

We take a brief moment to make a remark we became aware of through slides of Patrick Massot. If we were to define natural numbers via sets, as is sometimes taught, so that for example $0 \approx \emptyset$, $1 \approx \{\emptyset\}$ and $2 \approx \{\emptyset, \{\emptyset\}\}$, then statements such as $1 \in 2$ would be correct. This is another aspect of foundations that type theory resolves.

Next, we may define operations on natural number inductively, such as addition and subtraction, the latter of which requires defining predecessors of natural numbers first:

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

def nat.pred : ℕ → ℕ
| 0     := 0
| (a+1) := a

def nat.sub : ℕ → ℕ → ℕ
| a 0     := a
| a (b+1) := pred (sub a b)
```

Addition, for example, is defined inductively by recursing on the second term: if it is `nat.zero`, addition returns the first term, and if it starts with `nat.succ`, addition "passes the unit to the front". Thus, the way $2+2$ is computed is as follows: $2+2 = 2+(1+1) = ((2+1)+1) = (2+(0+1))+1) = ((2+0)+1)+1 = (2+1)+1$, and the latter is just notation for $(((0+1)+1)+1)+1$, for which 4 is syntactic sugar.

For subtraction, we get a first glance of the impact foundations can have. In Lean, $3 - 5$ evaluates to 0.
We have to introduce integers for $3 - 5 = -2$ to make sense. This "truncated" subtraction has the effect of often requiring special justification for lemmata we usually do not think of as having these requirements.
An example is: `nat.add_sub_assoc : ∀ {m k : ℕ}, k ≤ m → ∀ (n : ℕ), n + m - k = n + (m - k)`, the associativity of addition and subtraction for natural numbers, which now requires `k ≤ m`. Whereas in informal mathematics, we do not mind transitioning between integers and natural numbers, this is of great importance in formal mathematics, and requires additional justification.

Next, we will show how to define a property inductively, trough the example of $\leqslant$ for naturals:

```
inductive nat.less_than_or_equal (a : ℕ) : ℕ → Prop
| refl : less_than_or_equal a
| step : ∏ {b}, less_than_or_equal b → less_than_or_equal (succ b)
```

The constructors give us a blueprint for showing that $2 \leqslant 4$. With `step`, we can deduce $2 \leqslant 4$ from $2 \leqslant 3$. Now, to get $2 \leqslant 3$, we use `step` again, reducing the statement to be shown to $2 \leqslant 2$. This fact is given as axiom by the property, in the form of `refl` (for *refl*exivity). We may thus chain arguments to produce a proof:

```
example : 2 ≤ 4 :=
  nat.less_than_or_equal.step (nat.less_than_or_equal.step (nat.less_than_or_equal.refl))

example : 2 ≤ 4 :=
begin
  apply nat.less_than_or_equal.step,
  apply nat.less_than_or_equal.step,
  apply nat.less_than_or_equal.refl,
end
```

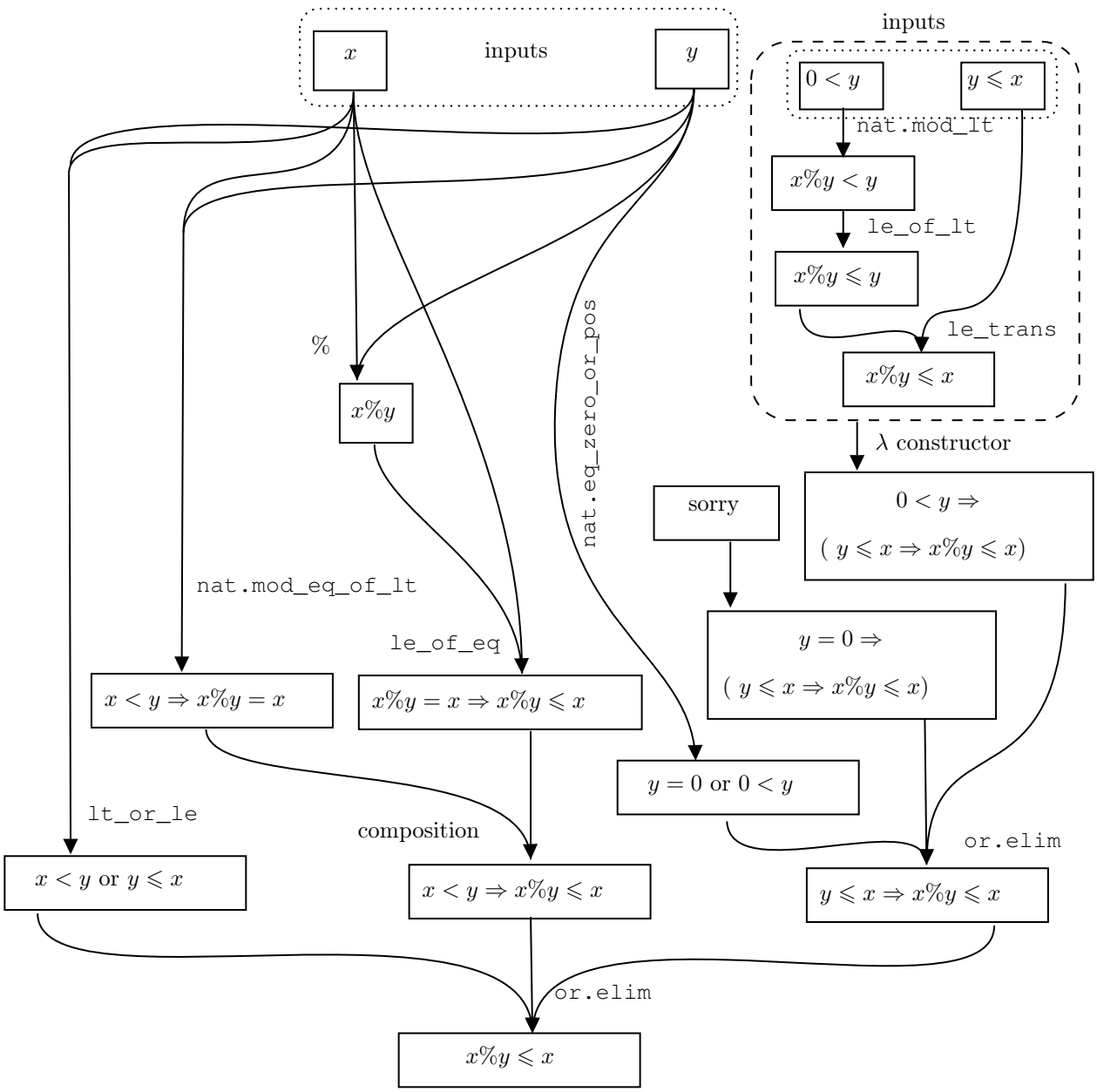
Note that `example` is just a way to declare nameless lemmata. We will explain the first syntax in a moment. The second syntax is known as a *tactic proof*. It is the preferred mode of writing proofs, as it allows for more interactivity.

This hints at what proofs are. They are "words" formed by composing functions. Giving a detailed explanation of what the latter statement means is beyond the scope of this thesis. We shall instead provide a more complex illustrative example:

```
example (x y : ℕ) : x % y ≤ x :=
  or.elim (lt_or_le x y)
  ((@le_of_eq ℕ _ (x % y) x) ∘ (@nat.mod_eq_of_lt x y))
  (or.elim (nat.eq_zero_or_pos y)
    ( (λ y0 ylex,
          (iff.elim_right ∘ (iff_of_eq ∘ (@congr_arg _ _ y 0 (λ y, x%y ≤ x))))
          (y0)
          (le_of_eq (nat.mod_zero x))))
    ((λ ypos ylex, le_trans (le_of_lt (nat.mod_lt x ypos)) ylex)))
```

This `example` states that for natural number x and y, the remainder in the Euclidean division of x by y is smaller or equal to x. This is lemma `nat.mod_le` from mathlib, and we used its proof as source of inspiration to produce the above one.

To see that this proof is a composition of functions, we have represented (most of) the proof as this diagram:



The content of the rectangular boxes are the objects or statements on them that make up the proof. The top layer represents the inputs x and y to the function that will constitute the proof of the `example`. The last box is the conclusion of the lemma we prove. When applying a function representing a lemma, we represent it as a directed-hyperedge, with its tails representing its inputs, and its head representing its output. The inputs are the objects and assumptions on them of the corresponding lemma, and the output is its conclusion. The name of the lemma is included near the head of the hyperedge.
The box with "sorry" contains the part of the proof we did not represent, for better readability.
Most hyperegdes represent mathlib lemmata. The edges labeled `%` and `composition` represent operation, namely taking the remainder, and composing functions, respectively. They are not fundamentally different from the lemmata: both are function ; the difference is that lemmata have property-types as outputs.
The λ-`constructor` allows us to build function from scratch. Consider the long-dotted box as a proof constructions of its own. It inherits inputs x and y of the main inputs, and adds hypothesis-inputs $0 < y$ and $y \leqslant x$, and will output `x % y ⩽ x`.

We encourage the reader to link this diagram to the previous code. We recall that function composition is such that the last function applied is the first to be read, when reading from the left. This corresponds to the bottom-most hyperedge of our diagram.

There is a different way to present this proof. This is the proof in the so-called *tactic mode*:

```
example (x y : ℕ) : x % y ≤ x :=
begin
  apply or.elim (lt_or_le x y),
  {intro x_lt_y,
   apply le_of_eq,
   apply nat.mod_eq_of_lt,
   exact x_lt_y,
   },
  {intro y_le_x,
   apply or.elim (nat.eq_zero_or_pos y),
   {intro y_0,
    apply @iff.elim_right _ (x % 0 ≤ x),
    apply iff_of_eq,
    apply @congr_arg _ _ y 0 (λ y, x%y ≤ x),
    exact y_0,
    --rw y_0,
    apply le_of_eq,
    exact nat.mod_zero x,
    },
   {intro y_pos,
    apply le_trans _ y_le_x,
    apply le_of_lt,
    apply nat.mod_lt,
    exact y_pos,
    },
   },
end
```

We encourage the reader to follow this proof by interacting with Lean.
By consulting the infowiew line by line, one will see how this proof "constructs", for lack of a better word, the previous diagram, piece by piece. If one replaces `example` with `lemma my_name`, and writes `#print my_name`, the infoview will display the proof in *term mode*, which is the composition of function we just discussed. The proof term of the above proof is very similar to the one we gave previously.

Tactic mode allows us to build proofs in a disorderly fashion, which is then assembled into the composition of functions that makes up the proof. We may reduce goals to sub-goals, using `apply`, which in the visual context of our previous diagram, corresponds to adding a hyperedge by its head to a node of the diagram connected to the initial goal. We may deduce intermediate results from the assumptions, using `have`, `replace`, or `specialise`, which in the visual context of our previous diagram, corresponds to adding a hyperedge by its tails to nodes of the diagram connected to the initial hypotheses.
The `λ-constructor` is built from the code blocks delimited by the syntax `{intro }`.
This example also illustrate how rewriting with `rw` relates to functions. The commented `rw y_0` can be used instead of the three preceding `apply` lines to achieve the same goal. Our example shows that rewriting is not a separate operation to composing functions.

Conclusion:
This concludes the exemplified overview of Lean's foundations we will give in this thesis. There is a lot more foundational material to be covered, and in much more depth. We will comment on the importance of understanding Lean's foundations in the next section.

3.4 Example of automation

Consider the following calculatory proof, were we show modifications to the proof state via comments:

```
lemma calculation_1
  (x y : ℚ)
  (h1 : x + y ≤ 1)
  (h2 : x - y ≤ -1) :
  x ≤ 0 :=
begin
  apply @le_of_mul_le_mul_left _ 2 x 0,
  /-
  ⊢ 2 * x ≤ 2 * 0
  ⊢ 0 < 2
  -/
  swap, norm_num, -- ⊢ 2 * x ≤ 2 * 0
  rw [two_mul, mul_zero], -- ⊢ x + x ≤ 0
  rw ← sub_self (y - 1), -- ⊢ x + x ≤ y - 1 - (y - 1)
  rw [sub_eq_add_neg, neg_sub], -- ⊢ x + x ≤ y - 1 + (1 - y)
  apply add_le_add,
  /-
  ⊢ x ≤ y - 1
  ⊢ x ≤ 1 - y
  -/
  {rw sub_eq_add_neg, -- ⊢ x ≤ y + -1
   rw ← sub_le_iff_le_add', -- ⊢ x - y ≤ -1
   exact h2 -- goals accomplished
   },
  {rw sub_eq_add_neg, -- ⊢ x ≤ 1 + -y
   rw ← sub_le_iff_le_add, -- ⊢ x - -y ≤ 1
   rw sub_eq_add_neg, -- ⊢ x + --y ≤ 1
   rw neg_neg, -- ⊢ x + y ≤ 1
   exact h1 -- goals accomplished
   },
end
```

It may be surprising to find such a long proof for such a basic calculation. However, going through the steps of the proof makes one realise how many operations and facts one implicitly uses for such calculations.
Fortunately, it turns out that such types of proofs can be automated:

```
lemma calculation_2
  (x y : ℚ)
  (h1 : x + y ≤ 1)
  (h2 : x - y ≤ -1) :
  x ≤ 0 :=
begin
  linarith,
end
```

Here, `linarith` is a so-called *tactic*, which is to say an algorithm that writes proofs.

We will not explain how `linarith` works, stating only that the algorithm underlying the computations for the proofs, is Fourier-Motzkin elimination. We stress that the latter is only the core of `linarith`: the difficulty lies in using an algorithm such as Fourier-Motzkin elimination to produce a proof using mathlib lemmata, such as the first one we presented above. Indeed, to check that `linarith` produces a proof, one may write `#print calculation_2` and see its proof term displayed in the infoview.

Hence `linarith` does not "cheat": we are still preducing proofs such as the first we presented in this subsection, except that we now do so algorithmically. The price to pay for automating proofs is that they potentially become longer and much less elementary. For example, one may compare the `linarith` proof term with that of the manual proof's one by writing `#print calculation_2`.

Writing tactics is referred to as *"metaprogramming"* (we write programs that when executed write other programs, hence the "meta"). In our `Introduction` file, we give an example of the code of a tactic. The code is relatively long and uninteresting for the scope of this thesis, so we do not reproduce it here.

The tactic is named `destruct_or`. It will prove disjunctions or propositions if one of the disjunction's terms is among the hypotheses. Successful applications are:

```
example (ha : a ∨ b):
  (d ∨ c) ∨ (a ∨ b) :=
  by {destruct_or,}

example (hb : b) :
  d ∨ ((d ∨ c) ∨ ((a ∨ b) ∨ (a ∨ a))):=
  by {destruct_or,}
```

The tactic `destruct_or` will branch over the disjunction through its parsing, and will at each step check if one of the hypothesese apply. We implement the branching via recursion. To clarify what we mean by "branching over the disjunction through its parsing", we recall that the logical `or`, $\vee$, is a binary operator, so that $a \vee b \vee c$ is either $a \vee (b \vee c)$ or $(a \vee b) \vee c$ depending on conventions. Then, "branching" corresponds to trying to prove each side. One may `#print` the latter proof and the following one, to see that they are the same.

```
example (hb : b) :
  d ∨ ((d ∨ c) ∨ ((a ∨ b) ∨ (a ∨ a))):=
  by {right,
      right,
      left,
      right,
      exact hb,}
```

What `destruct_or` did to find this proof is parsing the disjunction into `d` and `((d ∨ c) ∨ ((a ∨ b) ∨ (a ∨ a)))` and attempting to solve both with `b`. Seeing as this fails, it parses the left disjunction into `(d ∨ c)` and `((a ∨ b) ∨ (a ∨ a))`, and tries to solve them with `b`. This proceeds until it succeeds.

Tactics may fail, if incorrectly implemented or "misused". The following fails, for example:

```
example (ha : b ∨ a) :
  (d ∨ c) ∨ (a ∨ b) :=
  by {destruct_or,}
```

The reason is that `destruct_or` cannot handle commutativity of `or`.

Conclusion:
Certain proofs, or parts thereof, that allow for systematic/algorithmic solving, can be automated in Lean via tactics, the production of which is referred to as metaprogramming. Tactics greatly simplify the formalisation process. Finding the sort of statements that allow for a systematic proof, and designing an algorithm to perform these proofs is a active field of research/development.

4 The formalisation process

Before discussing concrete formalisation of content from "Proofs from THE BOOK" and its specificities, we discuss and exemplify aspects that are common to all formalisation projects. We then attempt to address possible misconceptions about formalisation. The code we refer to is listed in the file `FormalisationProcess` of our repository.

4.1 What is the formalisation process and what are its challenges ?

When formalising mathematics, we found ourselves moving repeatedly between 3 different tasks: elaborating the informal proofs/definitions, consulting mathlib for support, and actually writing the formal proof.

Elaboration
In the task of elaborating the informal proof/definition, one first fills in the informal details left out by authors of the source one wishes to formalise. This is nothing new to mathematician, but it is of particular importance in formalisation, as each step of a proof is essential in the formal proof. An example of this task we performed while formalising content from "Proofs from THE BOOK" is in the proof to the following theorem, contained in chapter 11 from our source. The implementation of this theorem can be found in the file `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry` of our repository.

Motzkin-Conway theorem

We consider a finite set X of at least 3 elements, and a family of m "abstract lines" A_i, that are proper subsets of X, so that for each pair of "points" of X, there is a unique A_i containing them.
Then, $|X| \leqslant m$.

Proof: We define r_x, the number of sets A_i that contain x, for a given $x \in X$.

- First we claim that $r_x < m$. We have $r_x \leqslant m$ by definition, and we shall show that $r_x = m$ is contradictory. Indeed, assume that for some x, all sets A_i contained x (which is what $r_x = m$ translates to). Seeing as X has at least 3 elements, we consider one more element a to x, and take note of the unique A_{i_1} that contains the pair $\{x, a\}$. This A_{i_1} is a proper subset of X by assumption, so there must exist a $b \in X$ such that $b \notin A_{i_1}$. Next, we record the unique A_{i_2} that contains the pair $\{x, b\}$, and the unique A_{i_3} that contains the pair $\{a, b\}$. Now, by assumption A_{i_3} must contain x. This means that A_{i_3} also contains pairs $\{x, a\}$ and $\{x, b\}$, so that by uniqueness $A_{i_1} = A_{i_2} = A_{i_3}$. This, however, implies that $b \in A_{i_1}$, which is the desired contradiction

Moving onward, we claim that if for some x and A_i, we have $x \notin A_i$, then $r_x \geqslant |A_i|$. To see this, note that for each element $y \in A_i$, there is some A_{i_y} containing $\{x, y\}$, each accounted for exactly once in r_x, as the A_{i_y} are distinct due to uniqueness, and contain x.
We may now start the proof. Assume for contradiction that $m < |X|$. Then, with our bounds, for any x and A_i such that $x \notin A_i$, we have $\frac{1}{m(|X| - |A_i|)} < \frac{1}{|X|(m - r_x)}$, where $|X| - |A_i|$ is the number of x such that $x \notin A_i$, and $m - r_x$ is the number of A_i such that $x \notin A_i$.
Thus, with some double-counting, we may derive the contraction $1 < 1$ from:

$$1 = \sum_{i=1}^{m} \frac{1}{m} = \sum_{i=1}^{m} \sum_{x : x \notin A_i} \frac{1}{m(|X| - |A_i|)} < \sum_{x \in X} \sum_{i : x \notin A_i} \frac{1}{|X|(m - r_x)} = \sum_{x \in X} \frac{1}{|X|} = 1$$

□

The paragraph that is initiated by the black dot contains details not present in the corresponding proof from "Proofs from THE BOOK". In this example, the detail left out by the source posed no threat to the formalisation project. Another example will be given in section 5.2 of this thesis, in which we fromalise a "proof by picture". However, one can envision cases in which certain details left out in a proof rely on fairly common, but non-elementary mathematics, such as a sophisticated inequality used in a calculation left out in the proof's exposition, for example. If such an inequality were not implemented in mathlib, then the workload that represents the formalisation project would increase drastically. Thus, to judge the workload of a formalisation project, it is necessary to first elaborate the informal details of one's source.

Mathlib
In a second phase, one thinks about what to formalise the content with, using the mathlib support one is aware of. That is, one determines the mathlib definitions and lemmata needed to perform the formalisation. These findings will guide the formalisation to come. Seeing as there is relatively little documentation for mathlib available, as of the time of this writing, the task of consulting mathlib for support is a non-trivial. Also, mathlib is modified over time, so depending on the mathlib version one is using, certain objects/lemmata may not be available, or have been modified. In a final step, one attempts to gain a practical understanding of the mathlib definitions and lemmata one recovered.

We discuss the task of finding the formal equivalents to informal objects/lemmata one wishes to use. Prior to November 2023, the best way to perform this task was to search through the mathlib folder by their names, or using the search engine of the API documentation [12]. The latter is not a keyword based search engine, so we found it rather unpractical to use, as one essentially has to know the names of the mathlib lemmata before using the search engine to find them. Now however, a keyword based search engine named Moogle [11] is publicly available. We did not make use of it, as we had produced our code by the time of its release, so we cannot account for its performance and its impact on the task of finding the appropriate objects/lemmata in mathlib.
Mathlib contains docstrings and preambles in its files that are supposed to guide the user on how to use the content of the file. Different mathlib authors will write documentation with different priorities and of different lengths. Some documentations are relatively detailed, such as that of mathlib file `combinatorics.hales_jewett`, and some are less detailed, such as `linear_algebra.span`. Mathlib for Lean 3 also contains a road-map folder, which contains relatively little content, focused on topology only, and is not present anymore in the mathlib of Lean 4. The latter contains a list of undergraduate mathematics concepts and their mathlib equivalent in `docs.undergrad`.
We conclude by noting that there is a continuous effort to ease the exploration of mathlib, but that this task is still relatively time consuming at the time of writing. We found that our currently best option for this task was to ask experienced Lean users online. The Zulip [13] stream "Is there code for X" is meant for this purpose. In section 5.1 of this thesis, we will see an example of the consequences that a flawed "reconnaissance" phase, that we just described, can have.

As we mentioned, there a few tutorials or road-maps to areas of mathlib currently available, and those that are available are not systematically listed on the official Lean website. For example, we found a tutorial on Lean's group theory at [2], and a tutorial on linear algebra at [3], which are both not present on the list [4], which is the Lean community's list for teaching resources. A possible reason for this lack of documentation is that mathlib is still in development, and its authors prefer to wait for a stable, community-wide accepted and richly illustrated stage of mathlib before writing documentation for it.
Some formal versions of mathematical object in mathlib are quite different from the usual way mainstream mathematicians think of them, such as mathlib's graphs, for example, and require a longer learning phase to be mastered. This raises the issue that once the formal equivalents of informal concepts necessary for the formalisation project have been retrieved, the user must first gain practical experience with the formalism. Tutorials or road-maps would greatly simplify this task.

We will elaborate example of graphs for the sake of concreteness. Mathlib's graphs are defined via relations on a type, where the type represents vertices, which are connected by an edge precisely when they are in relation.

```
structure simple_graph (V : Type u) :=
(adj : V → V → Prop)
(symm : symmetric adj . obviously)
(loopless : irreflexive adj . obviously)
```

Walks in graphs are defined inductively as follows:

```
inductive walk : V → V → Type u
| nil {u : V} : walk u u
| cons {u v w: V} (h : G.adj u v) (p : walk v w) : walk u w
```

Cycles are obtained from walks by first defining trails, which are walks without duplicate edges, then defining circuits, which are walks with the same start and end vertex, different from the 1-vertex walk, and finally by defining cycles to be circuits whose vertex set has no duplicates.

```
structure is_trail {u v : V} (p : G.walk u v) : Prop :=
(edges_nodup : p.edges.nodup)

structure is_circuit {u : V} (p : G.walk u u) extends to_trail : is_trail p : Prop :=
(ne_nil : p ≠ nil)

structure is_cycle {u : V} (p : G.walk u u)
  extends to_circuit : is_circuit p : Prop :=
(support_nodup : p.support.tail.nodup)
```

We will see these definitions used in practice in section 5.3 of this thesis. Matlib contains many ways to express the notion of subgraphs: there is the `subgraph` type, the relation `simple_graph.is_subgraph`, the coercions in from of `subgraph.coe` and `subgraph.spanning_coe`, as well as the notion of embeddings of graphs in form of `simple_graph.embedding`. In the case of cliques, there is also the `is_clique`, which is the notion we settled on when formalising an application of the previously stated Motzkin-Conway theorem to a theorem on the decomposition of a graph into cliques. The source for this theorem is in chapter 11 of "Proofs from THE BOOK" and its formalisation can be found in our file `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry`.
We now list these notions:

```
-- subgraph
structure subgraph {V : Type u} (G : simple_graph V) :=
(verts : set V)
(adj : V → V → Prop)
(adj_sub : ∀ {v w : V}, adj v w → G.adj v w)
(edge_vert : ∀ {v w : V}, adj v w → v ∈ verts)
(symm : symmetric adj . obviously)

-- simple_graph.is_subgraph
def is_subgraph (x y : simple_graph V) : Prop := ∀ v w : V, x.adj v w → y.adj v w

-- simple_graph.coe
def coe (G' : subgraph G) : simple_graph G'.verts :=
{ adj := λ v w, G'.adj v w,
  symm := λ v w h, G'.symm h,
  loopless := λ v h, loopless G v (G'.adj_sub h) }

-- simple_graph.spanning_coe
def spanning_coe (G' : subgraph G) : simple_graph V :=
{ adj := G'.adj,
  symm := G'.symm,
  loopless := λ v hv, G.loopless v (G'.adj_sub hv) }

-- simple_graph.embedding
abbreviation embedding := rel_embedding G.adj G'.adj
-- for context: rel_embedding
structure rel_embedding {α β : Type*} (r : α → α → Prop) (s : β → β → Prop) extends α ↪ β :=
(map_rel_iff' : ∀ {a b}, s (to_embedding a) (to_embedding b) ↔ r a b)

-- simple_graph.is_clique
abbreviation is_clique (s : set α) : Prop := s.pairwise G.adj
```

Note that this also exemplifies the issue of choosing the appropriate notion among multiple similar, informally equivalent ones, but formally different, ones in mathlib.

When we first encountered these definitions, we could not help but wonder why their mainstream equivalents were not implemented instead. Usually, mathematicians think of graphs as a set of vertices and a set of edges. We then tried to make our own graph library, in an attempt to imitate the mainstream way of thinking of graphs. We define graphs with:

```
structure newGraph (β : Type):=
(vertices : finset β )
(edges : finset (finset β ))
(is_graph : edges ⊆ vertices.powerset_len 2)
```

Here, we place vertices in a set of a given type and let edges be sets of vertices of size 2. We then define the notion of cycles in a graph via the existence of an embedding of a C_n in the graph, for some n, through the following notion of embedding. Here, f will be the embedding.

```
def is_graph_embed {α β : Type} [decidable_eq α] [decidable_eq β]
                   (H : newGraph α ) (G : newGraph β ) (f : α → β ): Prop :=
  (∀v ∈ H.vertices, f v ∈G.vertices ) ∧
  (∀v u, v∈ H.vertices → u ∈ H.vertices → (f v = f u) → (v = u) ) ∧
  (∀v u, v∈ H.vertices → u ∈ H.vertices →
        list.to_finset [u,v] ∈ H.edges → list.to_finset [f u, f v] ∈ G.edges)
```

With this version of graphs, in our file `TheBook_Pigeonhole_graphs_mine`, we prove the handshake lemma (named `handshake_newGraph`) and Reiman's theorem (named `max_edges_of_c4_free_Istvan_Reiman`), as a proof of practical usability of this formalism. In section 5.3, we will give a proof of the latter theorem using mathlib's graphs.

The formulation and application of this alternative definition of graphs illustrates that there are multiple ways to formalise certain mathematical notions in Lean. Some formalisms may be more practical then others. For example, in our personal formalism, we found that it would have been more efficient to define graphs via:

```
structure newGraph (β : Type):=
(vertices : finset β )
(edges : finset (finset β ))
(edge_subset_vert : edges ⊆ vertices)
(edge_size : ∀e, e∈ edges → e.card = 2)
```

Indeed, we found that we constantly had to rewrite the definition of `finset.powerset_len` with the previous definition, to obtain the facts we suggest in the above alternative definition. This is a example of an inefficient formulation.

We conclude that there may be considerable differences between the mainstream way of thinking of certain mathematical objects and their formal mathlib equivalents. We note that multiple formalisms are possible and that some are easier to work with then others. However, we feel obligated to mention that determining which formalism is best suited for mathlib does not seem to be an exact science. The authors of mathlib files will sometimes include a section on implementation details in the preamble of the files, but this type of documentation is rare and short.

As we noted, multiple formalisation approaches to a common notion may exist in mathlib. It may be unclear which of these notions to use. We have already described the case of subgraphs as an example of this problem. Other more substantial examples include:

- finset.prod and `finsupp.prod`, the first of which is fairly common, and the second of which is used in mathlib's notion of the decomposition into prime numbers, `nat.factorization_prod_pow_eq_self`.

```
def finset.prod [comm_monoid β] (s : finset α) (f : α → β) : β := (s.1.map f).prod

def prod [has_zero M] [comm_monoid N] (f : α →₀ M) (g : α → M → N) : N :=
∏ a in f.support, g a (f a)
```

- `set.Union`, `set.sUnion`, and `finset.bUnion`

```
def Union (s : ι → set β) : set β := supr s

def sUnion (S : set (set α)) : set α := Sup S

def bUnion (s : finset α) (t : α → finset β) : finset β :=
(s.1.bind (λ a, (t a).1)).to_finset
```

- On minute 03:20 from the talk available at [5], Kyle Miller says "there's sort of a battle going on about what should be `finite` and what should be `fintype`, but I'm hoping `finite` will win in the end".

```
class fintype (α : Type*) :=
(elems [] : finset α)
(complete : ∀ x : α, x ∈ elems)

class inductive finite (α : Sort*) : Prop
| intro {n : ℕ} : α ≃ fin n → finite
```

We called these examples "more substantial", as these are concepts mathematicians usually think of in a single way, unlike subgraphs, which may be thought of via subsets or embeddings, in the informal context too.

The coexistence of multiple formal notions for a single informal one is in principle not a problem, assuming that there is sufficient mathlib support for switching between these notions. Writing this support, and translating one's formalisation into all existing combinations of equivalents of objects/operation one is using, is however a time consuming task. Mathlib's policy encourages (and enforces, to some extent) a unified formalisation with few alternatives, as a consequence.

One more issue is that mathlib formalisations may become deprecated or outright deleted, or their names may be changed. Indeed, one may consult the `deprecated` folder to consult content the community no longer suggests using, and the change-log at [6]. This means that one must sometimes update one's mathlib knowledge.

Finally, we mention the fact that contributing to mathlib requires following relatively strict guidelines. The implementation of the Lovász local lemma available at [7], has been judged not idiomatic enough (it compiles, but it is not efficient/general/clean enough for the communities standards) and is hence not present on mathlib. For a reference of the criteria used when reviewing code that is to enter mathlib, one may consult [8]. It is our personal opinion that the code for the local lemma is well implemented and documented, and we would have liked to see it on mathlib. We also mention at this stage that there is no analogue to arxiv in Lean, in the sense that mathlib is an analogue to a journal. Repositories such as [7] are hence on no official list. In practice, to find out if a piece of math has been formalised, it is best to ask about it on the "Is there code for X" stream on Lean's Zulip. We mention at this stage, that we are not aware of an official system for keeping track of ongoing formalisation projects: we recommend asking if other users are already working on a topics on wishes to formalise on Lean's Zulip, before starting one's project.
For practitioners, this observation implies that the content that one wishes to formalise, or that may be a prerequisite to what one wishes to formalise, may have already been formalised, without being available on mathlib, nor available on a public list.

As an anecdote, we tried compiling the local lemma code from the above mentioned repository and received an unknown identifier error at `ennreal.mul_le_mul`. Using the change-log, we found that this lemma was deleted, with commit message (editor/reviewer justification) to be found at [9]. We then tried to implement the suggested changes, only to observe the following.

```
import tactic
import data.real.ennreal

example (a b c d : ennreal ): a ≤ b → c ≤ d → 0 ≤ c → 0 ≤ b → a * c ≤ b * d :=
  by {apply mul_le_mul,}
example (a b c d : ennreal ): a ≤ b → c ≤ d → 0 ≤ c → 0 ≤ b → a * c ≤ b * d :=
  by {apply @mul_le_mul ennreal a b c d _ _ _ _,}
```

The first example fails, whereas the second succeeds. We presume that the maintainer that implemented the deletion of `ennreal.mul_le_mul` believed the first syntax in the above examples to work, which would render `ennreal.mul_le_mul` obsolete. This illustrates the fact that mathlib is still in active development, and that from the practitioners viewpoint, it may be difficult to gain solid knowledge of an evolving library.

We will close our discussion of mathlib here, noting that it is a community project, hence the issues we discussed arise quite naturally and dissipate over time. Until then, one can count on Lean's welcoming and helpful community to answer one's question on Zulip or other platforms (Xena on Discord, in our case).

Formalisation
We found that the task of implementing the informal proof, given the knowledge of the mathlib equivalents to the concepts one is working with, is best performed by interacting with Lean. A good analogy would be that of of a puzzle: knowing roughly where pieces will be, one may sort them, and then piece them together relatively arbitrarily, where the decisions to try to join pieces together are either due to their similarity, or to the heuristic that is the knowledge of the final picture.
As we saw in section 3.2 of this thesis, formalisation may require one to make explicit steps one usually does not take under consideration in informal mathematics. In that example, the formal pigeonhole principle required us to provide an explicit map that sends the pigeons of $1, ..., 2n$ to pigeonholes among $\{1, 2\}, ..., \{2n - 1, 2n\}$. However, we found that informal mathematics is sufficiently founded and rigorous so that this aspect requires little creativity.

Writing the code
We have one last task of the formalisation process to discuss: actually writing the formal proof. The main reason formalisation consumes a large amount of time, besides the fact that formal proofs are naturally longer then their informal counterpart (they contain all details), is that even with a large mathlib culture, it is difficult to remember the names of lemmata needed to advance in the formal proof. When reviewing our code at the stage where we writing this thesis, we noted that we had already forgotten the meaning of some of the mathlib lemmata we used. Lean is a *language*, and as such, it must be constantly practiced so as to be properly memorised. Search tactics such as `library_search`, `suggest` and `find` exist, but are inefficient in practice. Indeed, they are not keyword based search engines, and are very sensitive to small changes or omissions in the search statement.
For example, the following will *not* find `nat.mod_lt`, which requires hypothesis $0 < y$ instead of $y \neq 0$.

```
example (x y : ℕ) (h : y ≠ 0) : x % y < y := by {library_search, }
```

We found that the currently fastest option to access mathlib lemmata is to try out different lemma names and watch the auto-complete's suggestions. The difficulty of searching for lemmata is probably the reason for mathlibs nomenclature for lemmata: for example, $a < b \to a \leq b$ is `le_of_lt`, which can be derived from "**l**ess-or-**e**qual is deduced **of** **l**ess, s**t**rictly". Indeed, mathlib lemmata names are descriptive. This is practical for short lemmata such as `le_of_lt`, but rather inconvenient for longer ones, such as `finset.exists_ne_map_eq_of_card_lt_of_maps_to`, for the pigeonhole principle.

Seeing as one has to search for lemmata names rather frequently, the additional minutes spent on this task can pile up to form entire workdays, over the course of a project. We will recall the keyword based search engine Moogle [11] and note that a continuous effort is made to make formalisation easier for users.

Detecting errors during formalisation
To conclude, we address the rather natural question of whether a mainstream mathematical result has been proven invalid in an attempt to formalise it. We found that all the content from "Proofs from THE BOOK" we formalised is formally correct. We can however point to the following testimony. In a talk by Bhavik Mehta, more precisely minute 26:07 of [1], an example from additive combinatorics is given, in which an additional assumption lacking in the informal statement was required, which lead to a increase in a multiplicative constant of the actual result.
As we mentioned in our introduction, Bhavik Mehta suggests the name "mathematical typos" for minor errors uncovered during formalisation, that do not greatly impact the statements of the formalised theorems.

4.2 What are possible misconceptions on formalisation?

Formalising recent publications
A perhaps common misconception is that formalisation in Lean has attained the stage where advanced mathematics can be formalised in a relatively short time period. As we hope to have communicated through the previous paragraphs, writing formal proofs is, for both natural and technical reasons, a much more time consuming endeavour then writing a journal article. We will discuss in more detail some criteria by which to tell if a source is difficult to formalise, which we list in order of increasing inducing difficulty:

- **Size:** Though counterexamples are plentiful, the size of a formal proof is roughly proportional to its informal counterpart. Hence, the longer the informal proof, the more time consuming the formalisation.
- **Interdisciplinarity:** If the source makes use of many areas of mathematics, then formalisation will require expertise in all these areas. Due to the precision required by formalisation, the user must posses both a solid understanding of the informal mathematical areas used, as well as a wide knowledge of mathlib's formalisation of these areas. We recommend working on interdisciplinary projects in a group of people, where each individual has specialised in a mathematical area.
- **Existence of mathlib support:** This is the most important factor. If the technical steps required by one's source have not been implemented in mathlib, one has to prove these steps first. Formalising the prerequisites of a source may sometimes be quite time consuming.

As of now, a considerable part of effort of formalising advanced results is developing basic mathematical content that is at the foundation of advanced results. For example, in our 4th proof of infinitude of primes, we had to formalise the notion of ordering a set of numbers and considering its kth element. As another example, we refer to the conclusion of the paper to the formalisation of the cap set problem [10] and draw attention to the statement "thousands more lines of general definitions and proofs were added to mathlib as part of this project" in its conclusion. Though this is a 2019 project, and much content has been produced since, some parts of mathlib are relatively underdeveloped. For instance, at the time of writing, mathlib does not contain the fact that trees have $|V| - 1$ edges, as can be checked on [14].

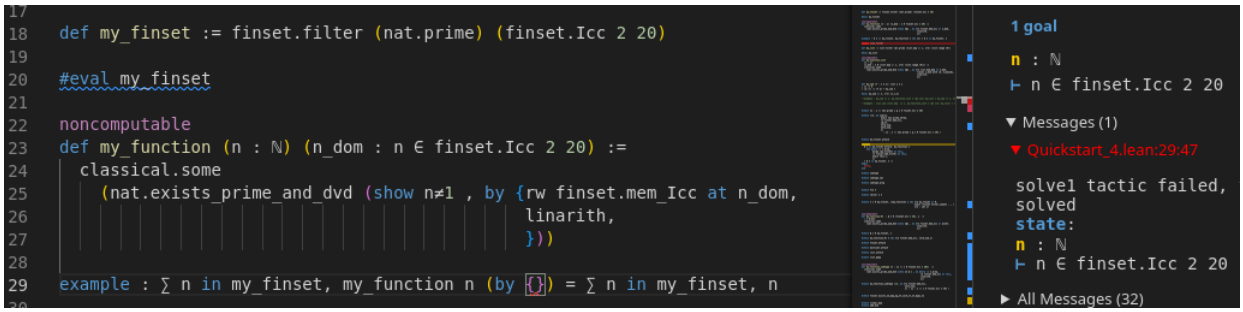
The importance of foundations
Another misconception we would like to address is that "one can avoid foundations", in the sense that it is not necessary to understand type theory or the mathematical foundations of objects in order to use them. We speak from personal experience when we make this recommendation. Throughout our work and the code that it produced, we worked with concepts such as coercion, inductive types, subtypes, pi-types, quotient types and instances of typeclasses. We found that a superficial understanding of these concepts leads to complications, errors and an outright halt in the formalisation process. To illustrate this point, consider:

```
def my_finset := finset.filter (nat.prime) (finset.Icc 2 20)

noncomputable
def my_function (n : ℕ) (n_dom : n ∈ finset.Icc 2 20) :=
  classical.some
    (nat.exists_prime_and_dvd (show n≠1 , by {rw finset.mem_Icc at n_dom,
                                              linarith,
                                              }))
```

```
example : ∑ n in my_finset, my_function n (by {}) = ∑ n in my_finset, n
```

The set `my_finset` contains the prime number between 2 and 20. The function `my_function` will take as inputs a number n and a proof that n is between 2 and 20, and output a prime divisor of it. When summing the images of `my_finset` under `my_function`, seeing as the only prime divisors of prime numbers are themselves, we expect this sum to simply be the sum of numbers in `my_finset`, as they are prime. this is the statement of the `example`.
Now, when attempting to fill in the second input to `my_function` with a proof that the input number is prime, in the `by {}`, we note the occurrence of the following:



Lean seems to have lost the information that the n that is being summed over is in `my_finset`. The reasons behind this phenomenon are complex, and we do not discuss them here. To solve this problem, one makes use of `finset.attach` which will produce a subtype that will carry the information that the n that is being summed over is in `my_finset`. This serves as an example for how technical type theory knowledge may be necessary to perform certain steps in a formalisation.

Another aspect that would also fall in this category of misconception is a good understanding mathlib's structures. Some lemmata, such as `mul_one_div_cancel` make use of handy structures such as `group_with_zero` so as to be easly used. Others however, do not, and this can result in some confusion. For example, there reason `prod_range_div` fails to rewrite $\prod_{i\in[n-1]} \frac{f(i+1)}{f(i)}$ to $\frac{f(n)}{f(0)}$, in:

```
lemma prod_range_telescope
  (f : ℕ → ℚ) (n : ℕ) (h : ∀ n : ℕ, f n ≠ 0) :
  ∏ k in range n, (f k.succ)/(f k) = (f n)/(f 0) :=
```

is because $\mathbb{Q}$ is not a commutative `group`, a condition required by the latter lemma (it is commutative additive group, `add_group`, which mathlib distinguishes from multiplicative groups so as to allow for easier work in fields). The latter type of problems may be relatively easy to diagnose, but they highlight the fact that if one wishes to write powerful code for mathlib, a good understanding of these structures is necessary.

Metaprogramming
Next, we recommend gaining some fundamental knowledge in metaprogramming and some understanding of technical aspects to Lean. This may potentially be deemed unnecessary from the "end-user"'s point of view, so we will elaborate on why we believe it to be useful. Lean is not flawless software. We have encountered erroneous error messages of form "term x has type y but is expected to have type y". Problems of this kind are not due to a flaw in the logical foundations of Lean, but rather to technical aspects, such as the naming of new variables or type inference (a sort of auto-completion). We were fortunate to have had access to the Xena's discord server community to help us out whenever we encountered such problems. If one wishes to gain some independence from the community when diagnosing these types of problems a good understanding of the technical aspects to Lean is necessary.

Mathlib's generality
Though it is not not exactly a misconception, we discuss a philosophy used for mathlib, which users may not be aware of, leading them to some confusion. This philosophy states that mathlib math should be as general (encompass as many cases and instances) as possible. This has certain effects that may be unexpected. For example, the fact that $a \leq b \to b \leq a \to a = b$ is not named `eq_of_le_of_le` in accordance to standard nomenclature,

but `le_antisymm`, since $\leq$ is an asymmetric relation, or the fact that finite unions of finite sets are described as `finset.sup _ id`, since finite sets may be ordered by inclusion, so that we may consider a supremum over them. As a rule of thumb, if one cannot find a concept that is elementary enough to be expected to be found in mathlib, then it is a good heuristic to look for a structure generalising it, which may be implemented instead.

A perhaps more anecdotic misconception related to the previous paragraph, is that formalisation in Lean is abstract on purpose. When formalising the Sylvester-Gallai theorem for this thesis (commented in section 5.2), the context we were working in was that of of a complete inner product space, despite our source working in $\mathbb{R}^2$. We did not make this choice with the ambition of generalisation. It simply represented less work to develop the theorem in the abstract context, as most of the support was formulated in the abstract context, so that specifying the support to $\mathbb{R}^2$ may have required additional work.

Idiomatic code

Finally, we address the idea that the first successful attempt at a formalisation is good enough. In the file `FormalisationProcess`, we included our first code for the 2nd proof of the infinitude of primes, with the last step contained in `InfinityViaFermat`. Though it may not seem so to the inexperience reader, and though the code compiles, this code is a bad as Lean code can get. Even the final code present in this thesis is at many parts not idiomatic. With our understanding, idiomatic code satisfies 3 main criteria: first, the code is general enough (for example, replace concrete instances such as $\mathbb{Z}$ by their abstract counterparts, such as a ring, in this case), next, the code is efficient (for example, term mode is preferred to tactic mode, and few costly tactics such as `linarith` are used), and finally, the code uses and generates mathlib support as much as possible (for example, it is preferred to partition proofs into their most elementary steps, each as its own lemma stated in a general form that makes it re-usable in other contexts). Unless one does not wish to participate to the development of mathlib, writing idiomatic code is a serious concern.
When reviewing our own code for this thesis, we realised that at many stages, the exposition could be simplified through one way or another. We believe that even with much formalisation experience, taking a second, close look at one's code is a good practice. In one's first formalisation effort, one is usually too concerned with checking that one's formalisation strategies work out, to consider the aspect of writing efficient code.

References:

1 https://www.youtube.com/watch?v=OMhBhbjR9S0

2 https://tqft.net/web/notes/load.php?name=students/20180219-MitchRowett-ASC-report-on-Lean

3 https://stavan-jain.github.io/linear_algebra_game/

4 https://leanprover-community.github.io/learn.html

5 https://www.slmath.org/summer-schools/1021/schedules/33445

6 https://mathlib-changelog.org/v3

7 https://github.com/nsglover/lean-lovasz-local-lemma/blob/main/src/lovasz_local_lemma.lean

8 https://leanprover-community.github.io/contribute/pr-review.html

9 https://github.com/leanprover-community/mathlib/commit/57ac39bd

10 https://doi.org/10.4230/LIPIcs.ITP.2019.15

11 https://www.moogle.ai/

12 https://leanprover-community.github.io/mathlib4_docs/

13 https://leanprover.zulipchat.com/

14 https://github.com/leanprover-community/mathlib4/blob/master/Mathlib/Combinatorics/SimpleGraph/Acyclic.lean

5 Case studies

We will now showcase concrete formalisations of content from "Proofs from THE BOOK". As we explained in the introduction, we chose to present the most challenging formalisations from our project, as these best illustrate the challenges of formalisation. We invite the reader to explore our repository to find the files we reference in this section, as well as the files we did not chose to include in our exposition. The files of our repository contain preambles that will provide directions to the reader to explore their content.
For convenience, we recall the link: https://github.com/Happyves/Master_Thesis

5.1 The 4th proof of infinitude of primes

We refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_4thProof`.

In chapter 1 of "Proofs from THE BOOK", 6 proofs of the infinitude of primes are given.

Infintude of primes:

There are infinitely many prime numbers.

Each of the 6 proofs shows this fact using different arguments or different areas of mathematics.
We shall recall the 4th proof, whose formalisation we will present in this section.
Proof: The proof may start with the following consideration: if P_n denotes the set of all primes $\leqslant n$, then we may consider the product of geometric series $\prod_{p\in P_n}\left(\sum_{i=0}^{\infty}\frac{1}{p^i}\right)$. When distributing this product of series, we obtain a series of terms of form $\prod_{p\in P_n}\frac{1}{p^{i_p}}$, for varying valuations i_p. Seeing as each number has a decomposition into primes, we may ask what numbers have their inverse in the latter series.
The number 1 to n surely are, as their prime factors must be smaller then themselves, hence smaller then n and therefore in P_n. This provides $1+\frac{1}{2}+...+\frac{1}{n}\leqslant\prod_{p\in P_n}\left(\sum_{i=0}^{\infty}\frac{1}{p^i}\right)$, where we see the harmonic series appear.
On the other hand. we may rewrite the geometric series $\sum_{i=0}^{\infty}\frac{1}{p^i}=\frac{1}{1-\frac{1}{p}}=\frac{p}{p-1}$,
so that $\prod_{p\in P_n}\left(\sum_{i=0}^{\infty}\frac{1}{p^i}\right)=\prod_{p\in P_n}\frac{p}{p-1}$.
We may actually upper-bound the latter product by a telescopic product.
To this end, we order the primes from 1 to $\pi(n)=|P_n|$, the number of primes $\leqslant n$, and use the bound $p_k\geqslant k+1$. This bound can be shown by induction: the first prime 2 satisfies $2\geqslant 1+1$, the second prime 3 satisfies $3\geqslant 2+1$, and for the step, note that the next prime is greater then the current prime's successor, so that $p_{k+1}\geqslant p_k+1\geqslant(k+1)+1$.
We may now derive $\frac{p_k}{p_k-1}\leqslant\frac{k+1}{k}$ by a quick computation, so that we may bound $\prod_{p\in P_n}\frac{p}{p-1}\leqslant\prod_{k=1}^{\pi(n)}\frac{k+1}{k}$.
The latter is a telescopic product, so that $\prod_{k=1}^{\pi(n)}\frac{k+1}{k}=\pi(n)+1$. Thus, we have $1+\frac{1}{2}+...+\frac{1}{n}\leqslant\pi(n)+1$.
We know from analysis that the harmonic series is unbounded, hence $\pi(n)$ is too.
Since $\pi(n)$ is the number of primes $\leqslant n$, there cannot be finitely many primes, as otherwise, their number would bound the harmonic series. □

As we shall see, the aspects important to formalisation may be unexpected to the reader. The distribution of a product of series may correctly be asserted as a formally difficult task with a low probability of existing support in mathlib. However, it turns out that ordering the prime numbers, so as to derive the bound $p_k\geqslant k+1$, as well as ordering a product of numbers is a non-trivial task that will require a serious formalisation effort.

Now, we shall dive into the formalisation. We defined our own prime counting function:

```
def π (n : ℕ) : ℕ :=
  ((range (n+1)).filter (λ p, nat.prime p)).card
```

This function is evaluable, as the line of code of the file that follow the definition show. For example, we may use Lean to compute that there are 9592 primes smaller then 100000. We later found out about mathlib's `nat.nth`, which allows one to define the nth natural number satisfying a given input property. This function is not evaluable, however, so our π is more satisfactory.

Our first concern when formalising this proof was with ordering the primes. This is a crucial aspect of the proof, as the bound $p_k \geqslant k+1$ on the kth prime required primes being ordered.
Mathlib provides `finset.sort`, which, given a finite set, will produce a list (a data-structure of Lean) of the ordered elements of the finite set. We use it to define the kth prime among the first n ones:

```
def kth_prime_among (n k: ℕ) (h : k < (π n)) :=
  list.nth_le (finset.sort (≤) ((range (n+1)).filter (λ p, nat.prime p))) k
  (by {rw [π] at h, simp only [finset.length_sort], exact h, })
```

The fact that we provide the upper-bound n makes this function evaluable, and we provide examples of computations in the lines that follow the definition in the referenced file. We use `list.nth_le`, an algorithm on lists that returns the kth element of the list, provided we show that k does not't exceed the length of the list. The latter is the reason we require hypothesis `h` as input to our function.

The informal proof also requires ordering a product: $\displaystyle\prod_{p\in P_n} \frac{p}{p-1} = \prod_{k=1}^{\pi(n)} \frac{p_k}{p_k-1}$.
Anticipating the need for a bijection to permute the terms of this product, we defined the "rank" of a prime in a specified range. Again, specifying the range will make the function evaluable. We used the algorithm `list.index_of` to define the function, as an equivalent for finite set does not exist in mathlib.

```
def prime_rank_among (n p: ℕ) (h : p ∈ ((range (n+1)).filter (λ q, nat.prime q))) : fin (π n) :=
  ⟨ list.index_of p (finset.sort (≤) (((range (n+1)).filter (λ q, nat.prime q)))),
    by {simp only [π], rw ← finset.length_sort has_le.le,
        rw list.index_of_lt_length, rw finset.mem_sort, exact h,}⟩
```

The output of this function has type `fin (π n)`, the type of natural numbers strictly less then $\pi(n)$. Lean allows to concatenate a type of objects and a crucial property tied to in into a single type. This is done quite frequently, even for usual mathematical objects, in mathlib.
Here, we used `fin (π n)` so as to give aesthetic formulations of the fact that the functions `kth_prime_among` and `prime_rank_among` are inverses of one another, two facts recorded in lemmata `order_tec_1` and `order_tec_2`.

```
lemma order_tec_1
  (n p: ℕ) (h :p ∈ ((range (n+1)).filter (λ q, nat.prime q))):
  kth_prime_among n (prime_rank_among n p h).val (prime_rank_among n p h).prop = p :=

lemma order_tec_2
  (n k: ℕ) (h : k < (π n)) :
  prime_rank_among n (kth_prime_among n k h) (kth_prime_among_makes_sense n k h) = ⟨k,h⟩ :=
```

Had we not used `fin (π n)` as output type, then we would have had to prove that `prime_rank_among` has values ranging from 0 to $\pi(n)-1$ in a separate lemma, and mention that lemma each time we need this fact, where as we may use syntax `.prop` to access that fact when using `fin (π n)`. Needless to say, this choice is arbitrary and not essential to the functioning of the definitions.

As a last remark on `prime_rank_among`, we note that it starts counting primes from 0, whereas in the informal proof, we started counting at 1. This will affect the bound $p_k \geqslant k+1$ on the kth prime later on.

We may then successfully carry out $\prod_{p \in P_n} \frac{p}{p-1} = \prod_{k=1}^{\pi(n)} \frac{p_k}{p_k - 1}$ in lemma `order_the_prod` of the file.
The equality is show using mathlib lemma:

Rewriting products
`finset.prod_image`

Given an injective map g, we may rewrite $\prod_{x \in g(s)} f(x) = \prod_{x \in s} f(g(x))$.
In code: `∀ f : α → β s : finset γ g : γ → α, (∀ (x : γ), x ∈ s → ∀ (y : γ), y ∈ s → g x = g y → x = y) → ∏ (x : α) in image g s, f x = ∏ (x : γ) in s, f (g x)`

Here, s will be $[\pi(n)]$ (more precisely $0, ..., \pi(n)-1$, though we stick to the previous enumeration as it fits the informal exposition), g will be `kth_prime_among`, and f will be $x \mapsto \frac{x}{x-1}$. This requires us to show that $P_n = g([\pi(n)])$, which we do in our lemma `tec_order_set`

We then move on to prove the bound $p_k \geqslant k+1$, which is our `kth_prime_among_bound`: by starting to count at 0, we must actually show $p_k \geqslant k+2$. The bound is shown by induction, just as in the informal proof. The difficulty was in justifying the fact that "the next prime is greater the the current" on, which in our context translates to `kth_prime_among` being strictly increasing, which is shown in `kth_prime_among_increase`. It required us to prove two technical lemmata about the algorithm `list.nth_le`, that we could not find in mathlib, and named `list_stuff_2` and `list_stuff_3`.

```
lemma kth_prime_among_bound (n k: ℕ) (h : k < (π n)) : k+2 ≤ kth_prime_among n k h :=

lemma list_stuff_2
  (l : list nat) ( hl : list.sorted (≤) l) (k : nat) (hk : k.succ < l.length) :
  l.nth_le k (by {rw nat.succ_eq_add_one at hk, linarith,}) ≤ l.nth_le k.succ hk :=

lemma list_stuff_3
  (l : list nat) ( hl : list.nodup l) (k : nat) (hk : k.succ < l.length) :
  l.nth_le k (by {rw nat.succ_eq_add_one at hk, linarith,}) ≠ l.nth_le k.succ hk :=
```

Next, we show $\prod_{k=1}^{\pi(n)} \frac{p_k}{p_k - 1} \leqslant \prod_{k=1}^{\pi(n)} \frac{k+1}{k}$ in our lemma `prod_ordered_primes_bound_pre`. We have included the algebraic manipulations providing $\frac{p_k}{p_k - 1} \leqslant \frac{k+1}{k}$ in the proof of this lemma, which is why its proof appears bigger then need be.

We then reduce the telescopic product in `prod_ordered_primes_bound`: $\prod_{k=1}^{\pi(n)} \frac{p_k}{p_k - 1} \leqslant \pi(n) + 1$
We experienced some formal difficulties proving this. Mathlib provides `finset.prod_range_div` to telescope in products. This requires the product to be over a commutative multiplicative group, which the $\mathbb{Q}$ we are working in is not. Hence we develop our own version: `prod_range_telescope`.

Informally speaking, we have arrived the inequality $\prod_{p \in P_n} \frac{p}{p-1} \leqslant \pi(n) + 1$ at the current stage.
We now move to the second difficulty of this proof: lower-bounding the product by the harmonic series.

If one recalls the informal proof, the goal would now be to show that $\prod_{k=1}^{\pi(n)} \frac{p_k}{p_k - 1} = \prod_{p \in P_n} \left(\sum_{i=0}^{\infty} \frac{1}{p^i} \right)$.
This would require us to use series. The closest mathlib material to a theory of series we could find can be found in the files `analysis.p_series` and `analysis.analytic.basic` of mathlib. We could not find the crucial operation of developing the product of these series. Also, learning a new part of mathlib is quite time consuming, so we decided to modify the informal proof, staying in the familiar universe of finiteness instead.

We now present the informal work-around we will develop from now on.
Seeing as we need lower bounds only, we make use of finite geometric sums, through bounds:

$$\sum_{i=0}^{n} \frac{1}{p^i} = \frac{1 - \left(\frac{1}{p}\right)^{n+1}}{1 - \frac{1}{p}} \leqslant \frac{1}{1 - \frac{1}{p}} = \frac{p}{p-1}$$

This reduces the problem to showing $1 + \frac{1}{2} + ... + \frac{1}{n} \leqslant \prod_{p \in P_n} \left(\sum_{i=0}^{n} \frac{1}{p^i} \right)$.
We may now safely distribute $\prod_{p \in P_n} \left(\sum_{i=0}^{n} \frac{1}{p^i} \right) = \sum_{v : P_n \to [0,n]} \prod_{p \in P_n} \frac{1}{p^{v(p)}}$, where the sum ranges of all functions mapping primes of P_n to a number from 0 to n. Seeing as terms are positive, we may reach the lower-bound by the harmonic by showing that $1, \frac{1}{2}, ..., \frac{1}{n}$ may be written in form $\prod_{p \in P_n} \frac{1}{p^{v(p)}}$, for some $v : P_n \to [0, n]$, which will correspond to valuations in the prime decomposition.

This last bounding-step requires some formal attention. We are comparing sums $\sum_{i \in [n]} \frac{1}{i}$ and $\sum_{v : P_n \to [0,n]} \frac{1}{\prod_{p \in P_n} p^{v(p)}}$.
There are two arguments that come to mind for making the comparison: either, we display an injective map sending all $\frac{1}{i}$ to terms of the form $\frac{1}{\prod_{p \in P_n} p^{v(p)}}$, for some $v : P_n \to [0, n]$, or we display a surjective map sending all $\frac{1}{\prod_{p \in P_n} p^{v(p)}}$ to terms of form $\frac{1}{i}$, for $i \in [n]$.
If we chose the first option, using the decomposition into primes for map, showing its injectivity would require showing uniqueness of the decomposition into primes.

This is not necessary for the second option, which in turn has its own problem. Indeed, how would we match terms $\frac{1}{\prod_{p \in P_n} p^{v(p)}}$, for v ranging over all $P_n \to [0, n]$, with terms of form $\frac{1}{i}$, for $i \in [n]$?
To circumvent this formalisation problem, we will prove two intermediate bounds:

$$\sum_{i \in [n]} \frac{1}{i} \leqslant \sum_{\substack{v : P_n \to [0,n], \\ \prod_{p \in P_n} p^{v(p)} \in [n]}} \frac{1}{\prod_{p \in P_n} p^{v(p)}} \leqslant \sum_{v : P_n \to [0,n]} \frac{1}{\prod_{p \in P_n} p^{v(p)}}$$

To show the first bound, we may define our surjection to be $v \mapsto \prod_{p \in P_n} p^{v(p)}$. To show the second, we use the facts that terms are positive and that we are summing over a subset.

We shall now translate these ideas into code. We make the transition to finite geometric sums in `prod_sum_bound`. Distributing the product of sums, in `the_great_split_part_1`, is achieved with:

Distribution of products and sums
`finset.prod_sum`

We may distribute along the pattern $\prod_{i\in S}\left(\sum_{j\in T_i} f(i,j)\right) = \sum_{\substack{m : S\to \cup_i T_i \\ m(i)\in T_i}} \prod_{i\in S} f(i,m(i))$.

In code:
```
∏ (a : α) in s, ∑ (b : δ a) in t a, f a b = ∑ (p : ∏ (a : α), a ∈ s → δ a) in s.pi t,
∏ (x : {x // x ∈ s}) in s.attach, f x.val (p x.val _)
```

In `the_great_split_part_2`, we show $\sum_{\substack{v : P_n\to[0,n], \\ \prod_{p\in P_n} p^{v(p)}\in[n]}} \frac{1}{\prod_{p\in P_n} p^{v(p)}} \leqslant \sum_{v:P_n\to[0,n]} \frac{1}{\prod_{p\in P_n} p^{v(p)}}$.
Its main arguments are encapsulated in mathlib's `finset.sum_le_sum_of_subset_of_nonneg`, which states that the sum over a subset of the index set, for non-negative terms, is smaller the the initial sum.

We peak ahead in our file to our lemma `sum_nonneg_surj`.

```
lemma sum_nonneg_surj
  {α β γ: Type} [decidable_eq α] [decidable_eq γ] [ordered_add_comm_monoid β]
  {s : finset α} {t : finset γ} {f : α → β} {g : γ → β}
  (i : ∏ a ∈ s, γ) (hi : ∀ a ha, i a ha ∈ t) (h : ∀ a ha, f a = g (i a ha))
  (i_surj : ∀ b ∈ t, ∃ a ha, b = i a ha) (nonneg_fun: ∀ a, a∈s → 0 ≤ f a):
  (∑ x in t, g x) ≤ (∑ x in s, f x) :=
```

It will allow us to show $\sum_{i\in[n]} \frac{1}{i} \leqslant \sum_{\substack{v : P_n\to[0,n], \\ \prod_{p\in P_n} p^{v(p)}\in[n]}} \frac{1}{\prod_{p\in P_n} p^{v(p)}}$ by constructing a surjective map, namely $v \mapsto \prod_{p\in P_n} p^{v(p)}$, from $\left\{v : P_n\to[0,n] \mid \prod_{p\in P_n} p^{v(p)}\in[n]\right\}$ to $[n]$, which maps terms of the right hand-side sum to terms of the left hand-side sum of the previous inequality.

As we hinted at earlier, surjectivity follows from the existence of a decomposition into primes for natural numbers. Mathlib has a computational notion of prime decomposition in the form of `nat.factorization_prod_pow_eq_self`. For example, one may write `#eval nat.factorization 2023 7` to have Lean compute that the valuation of 7 in the prime decomposition of 2023 is 1. Prime decomposition is present in the from of `nat.factorization_prod_pow_eq_self`, which makes use of finitely supported functions, and their sums. The transition to finset-sums is achieved with `finsupp.prod_of_support_subset`. We did not know of the existence of these theorems when formalising our content. We were only aware of a different version of prime factorisation `nat.prod_factors`, in which valuations made no appearance. This lead us to show our own version of prime decomposition `quick_prime_decompo`. It is shown by strong induction, where we make use of mathlib's `nat.exists_prime_and_dvd` to obtain prime divisors in the induction step.

```
lemma quick_prime_decompo {n : ℕ} (m: ℕ) (mdef: m ∈ Icc 1 n) :
  ∃ (valu : ∏ (valu : ℕ), valu ∈ filter (λ (p : ℕ), nat.prime p) (range (n + 1)) → ℕ),
  m = ∏ (x : {x // x ∈ filter (λ (p : ℕ), nat.prime p) (range (n + 1))}) in (filter (λ (p : ℕ),
      nat.prime p) (range (n + 1))).attach, ↑x ^ valu ↑x x.prop :=
```

Finally, we may carry out our proof strategy, in the proof of `the_great_split_part_3` in which the harmonic series first makes its first appearance as a lower bound. We may then merge all the bound to finally lower-bound the prime counting function π by the harmonic series, in our theorem `the_great_merger`.

In "Proofs from THE BOOK", the harmonic series is lower-bounded by the logarithm through a visual argument. The divergence of the prime counting function π is then derived from the divergence of the logarithm.
We could not find the bound with the logarithm to the harmonic series in mathlibs analysis folder. The visual argument used in our source to derive this bound is currently practically inaccessible in Lean. To our knowledge, the notions of area and integration are not developed enough to tackle this type of argument.
It is however shown in mathlib's `real.tendsto_sum_range_one_div_nat_succ_at_top` (in `analysis.p_series`, that the harmonic series diverges. The proof is based on the Cauchy condensation test.
We decided to circumvent mathlib's analysis formalisation once more, and gave a more elementary proof of the harmonic series divergence in lemmata `harmonic_lb` and `harmonic_unbounded` of our file. We give its informal version here:

Divergence of the harmonic series

There is no bound b such that for all n, we have $\sum_{i\in[n]} \frac{1}{i} \leqslant b$. More precisely, we have $\frac{n}{2} \leqslant \sum_{i\in[2^n]} \frac{1}{i}$.

Proof: The latter statement implies the first: if such a bound b existed, then evaluating the latter bound in $n = 2b + 2$ would yield $b + 1 \leqslant b$. So it remains to prove the second statement.
It is shown by induction. The base case is $0 \leqslant 1 = \sum_{i\in[2^0]} \frac{1}{i}$.
For the step, note that $\sum_{i\in[2^{n+1}]} \frac{1}{i} = \sum_{i\in[2^n]} \frac{1}{i} + \sum_{i\in[2^n+1,\ 2^{n+1}]} \frac{1}{i} \geqslant \frac{n}{2} + \sum_{i\in[2^n+1,\ 2^{n+1}]} \frac{1}{2^{n+1}} = \frac{n+1}{2}$. □

Formally, we show:

```
lemma harmonic_unbounded : ¬ (∃ b : ℕ, ∀ n : ℕ, ∑ k in Icc 1 n, (1/k : ℚ) < b) :=
```

We may then finally conclude with the infinitude of primes in `fourth_proof`.

```
theorem fourth_proof : {n : ℕ | n.prime }.infinite :=
```

The proof is by contradiction. If there were finitely may primes, say p, then we would have $\pi(n) + 1 \leqslant$, so that $b = p + 1$ would serve as upper-bound to the harmonic series, which we've just proven cannot exist.

This concludes our formalisation of the 4th proof of the infinitude of primes.
We would like to point out that this proof may possibly be shortened at certain parts, by tying it into currently existening, and possible future support from mathlib.
Making use of `nat.factorization_prod_pow_eq_self` and of `real.tendsto_sum_range_one_div_nat_succ_at_top`, provided the transition to our context (finite sums instead of `finsupp.sum`, unboundedness instead of limits) is well supported in mathlib, would spare us having to show our own versions of these results.
It is also possible that support for ordering elements of finsets, and refering to the kth element of a finset, will be developed in mathlib, so that our `kth_prime_among` and its properties can be derived from a more general context.

5.2 The Sylvester-Gallai theorem

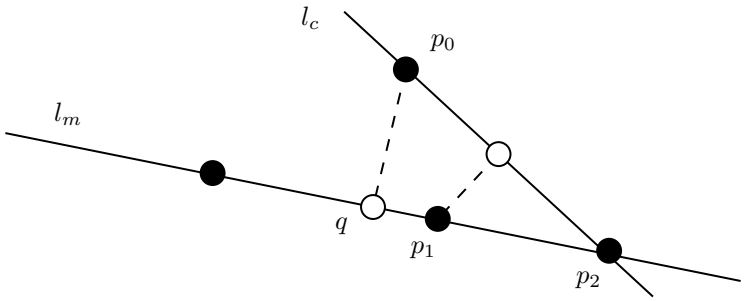
We refer to file `FormalBook_Ch11_LinesInThePlane_SylvesterGallai`.

We want to formalise the following theorem, from chapter 11 from "Proofs from THE BOOK", which is about incidence geometry and an application of it to graph theory.

Sylvester-Gallai theorem

For n points, not all on the same line, there is a line containing exactly 2 of the points.

Proof: We denote the set of n points by P, and define the set of lines L passing through at least 2 points of P. For each point $p \in P$ and each line $l \in L$, such that $p \notin l$, we may consider the distance of p to l (the distance of the p to its orthogonal projection on l, or equivalently the shortest distance from p to any point on l).
We consider the pair p_0 and l_m for which this distance is minimised, and show that l_m is the line we seek: it contains exactly 2 points of P. We show the latter by contradiction, assuming that l_m has at least 3 points of P on it.
If q denotes the orthogonal projection of p_0 on l_m, then among the (at least) 3 points of P, two must be on the same side of the line, when we split the latter along q. We denote them by p_1 and p_2, so that we may represent the situation as in the following figure.



As represented in the figure, we next consider the line $l_c \in L$ defined by the points p_0 and p_2.
Then, the distance between p_1 and l_c is smaller then that between p_0 and l_m, which contradicts the minimality of the latter distance among the pairs of P and L. □

The theorem has a geometric context. The probably most adequate formal context to frame it in is that of affine spaces. Mathlib has support for these, in the form of files `linear_algebra.affine_space.affine_subspace`, which would lead us to express lines as one-dimensional affine spaces first, and `geometry.euclidean.basic`, which contains support for orthogonal projections on affine spaces. However, we could not find a the characterisation of the affine projection as a distance minimiser in the latter file.
When setting out to formalise the above proof, we made use of this characterisation in a step that will elaborate the fact that distance between p_1 and l_c is smaller then that between p_0 and l_m. We therefore decided to shift the context to that of inner product spaces, since the mathlib file `analysis.inner_product_space.projection` contains the desired characterisation, in the form of `exists_norm_eq_infi_of_complete_convex`.

In this context, we define lines as follows:

```
def line (a b : E) : set E := {x : E | ∃ t : ℝ, x = a + t•(b-a) }
```

Note that these may be points if $a = b$.

We then proceed to show that lines are nonempty, in our `line_nonempty`, complete, in our `line_complete`, and convex, in our `line_convex`. These conditions are necessary for us to define the orthogonal projection of a point on such a line, using mathlib's `exists_norm_eq_infi_of_complete_convex`. We did not finish the proof that lines are complete, a task that would have required the time costly endeavour of learning a new part (topology) of mathlib.

We then show some rewrite-lemmata, that allow us to express lines in terms of the points they contain, and that we'll use at various stages throughout the file(s). Their proofs serve as good examples for algebraic manipulation in vector spaces.

```
lemma left_mem_line {a b : E} : a ∈ line a b :=

lemma line_comm {a b : E} :  line a b = line b a :=

lemma right_mem_line {a b : E} :  b ∈ line a b :=

lemma line_rw_of_mem  {a b c : E} (h : c ∈ line a b) :  line a b = line c (c+(b-a)) :=

lemma line_rw_of_mem_of_mem  {a b c d: E}  (hcd : c ≠ d)
       (hc : c ∈ line a b) (hd : d ∈ line a b):  line a b = line c d :=
```

We first set out to show that for 4 points on a line, when splitting the line along one of the points, then among the remaining 3, 2 of them are on the same side of the split. This is our lemma `pigeons_on_a_line`. In the proof of Sylvester-Gallai, it will be used to obtain the points p_1 and p_2 that are on the same side when splitting l_m along q. It states as follows, where the 4 points are denoted a, b, c, p, and we split the line along p :

```
lemma pigeons_on_a_line
  {a b c p : E }
  (hc : c ∈ line a b) (hp : p ∈ line a b)
  (hab : a ≠ b) (hac : a ≠ c) (hcb : c ≠ b) :
  ∃ x, (x = a ∨ x = b ∨ x = c) ∧
  ∃ y, (y = a ∨ y = b ∨ y = c) ∧
  (y ≠ x) ∧
  ∃ t : ℝ, (0 < t ∧ t ≤ 1) ∧
  y = x + t•(p-x) :=
```

In the goal, x and y are the points on the same side, and y will be the point between x and p. With respect to our previous figure, x is p_2, y is p_1 and p is q. Seeing as we must account for all possible cases of the configuration of the triple of points on the line, we have the disjunctions on values of x and y.
As the name we gave the lemma suggests, this is proven with the pigeonhole principle. Points on the line can be expressed as $p + t(b - a)$ for some scalar t, where $b - a$ is the direction of the line. The sign of this scalar (≥ 0 or < 0) determines two sides of the line. Seeing as we have 3 points a, b, c, the pigeonhole principle guarantees that 2 points will be on the same side.
We will not discuss the proof of `pigeons_on_a_line` so as to save time, though we encourage the reader to look it up in the referenced file to judge its considerable length.

Next, we define the pairs of points and lines, that we will consider the distances of:

```
def point_line_finset (P : finset E) :=
  (finset.univ : finset (P × (P × P))).filter
    (λ t, (t.1 ≠ t.2.1) ∧
          (t.1 ≠ t.2.2) ∧
          (t.2.1 ≠ t.2.2) ∧
          (t.1.val ∉ line t.2.1.val t.2.2.val))
```

Seeing as our lines are defined by a pair of points, we represent a pair of a point and a line by a triple of points. The main condition, which is that the point is not on the line, is present in form of the last condition we filter on, in the above definition. Here, the points have type P, where we use the type to carry information, and to be able to stay in the context of finsets, as Lean recognises that `P × (P × P)` is a finite type, and hence we may use the notion of `finset.univ` for it, the universal *finite* set of triples of points.

Next, we move to our lemma `point_line_finset_nonempty`. Here, we show that if the points of P are not aligned, the the set of point-line pairs we just defined, `point_line_finset` is non-empty. We will need this technical fact to define the minimum distance among the distances of points to lines of the point-line pairs, using `finset.min'`.

For a given point-line pair, we now define the orthogonal projection of the point on that line:

```
def point_line_proj
  (P : finset E) (hSG : ¬ (∃ a b : E, ∀ p ∈ P, p ∈ line a b))
  (t : (P × (P × P))) :=
classical.some
  (@exists_norm_eq_infi_of_complete_convex _ _ _
   (line t.2.1.val t.2.2.val)
   (by {apply line_nonempty,})
   (by {apply line_complete,})
   (by {apply line_convex,})
   t.1.val)
```

Here, we make use of the following key existence result, for which we could find no equivalent in the affine spaces part of mathlib, and which thereby lead us to develop the theorem in the context of complete inner product spaces.

Existence of a norm minimiser
`exists_norm_eq_infi_of_complete_convex`

For a nonempty, complete, convex set K, and a point u, we may find a point $v \in K$ that minimises the distance from u to the points of K.
In code: `∀ (u : F), ∃ (v : F) (H : v ∈ K), ∥u - v∥ = ⊓ (w : K), ∥u - ↑w∥`

The symbol ⊓ denotes one of Leans notions of infinimum. Here, K is the line spanned by the last two point of the triple, and u is the first point of the triple. To be able to speak of one of the minimisers, we makes use of the following, which is related to the axiom of choice:

Exsitencial elimination
`classical.some`
`classical.some_spec`

If we know that something of type T exists, which satisfies predicate p, then we may refer to it with `classical.some`, and use `classical.some_spec` to certify that it satisfies the predicate p.
In code, respectively:

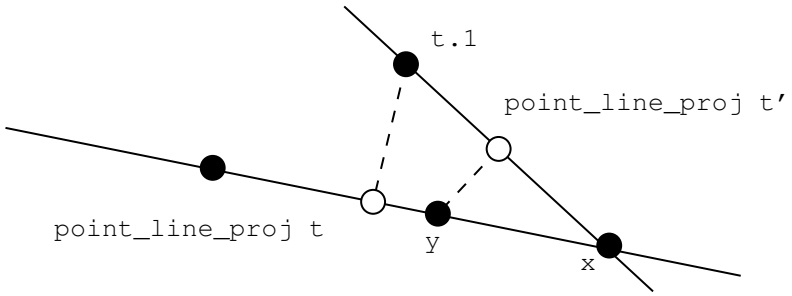
- `∏ {α : Sort u} {p : α → Prop}, (∃ (x : α), p x) → α`
- `∀ {α : Type} {p : α → Prop} (h : ∃ (x : α), p x), p (classical.some h)`

We use `classical.some_spec` to collect the property of our projection as a minimiser in our lemma `point_line_dist_def`.

We may now define the minimum distance among pairs of points and lines. To do so, we consider the image of our set of triples (representing point-line pairs) under the map that associates the triple to the distance between the first point and its projection on the line spanned by the other two.
We then consider the minimum of that image.

```
def min_dist
  (P : finset E) (hSG : ¬ (∃ a b : E, ∀ p ∈ P, p ∈ line a b)) :=
  finset.min'
    (finset.image
      (λ t : P × P × P, ‖↑t.1 - (point_line_proj P hSG t)‖ )
      (point_line_finset P))
    (by {apply finset.nonempty.image,
         exact point_line_finset_nonempty P hSG,})
```

We have, a priori, established all notions relevant to the proof of the Sylvester-Gallai theorem. In our file, we skip ahead to out formalisation of the theorem, in `Sylvester_Gallai`. In its proof, we let d be the minimum distance between point-line pairs. We consider the triple of points t that it corresponds to, and use the line spanned by the two last points of the triple as candidate for the points spanning a line that contains only 2 points of P.
Next, we prove the property of the line by contradiction, assuming that there is a third point q on that line.
We order the points on the line withe the help of `pigeons_on_a_line`, so that x and y are the points on the same side of the line, when split along the projection of the first point of t on the line.
We then define the triple $t' = (y, x, t.1)$, where $t.1$ is the first point of triple t.
With our notation, we have reached the following situation in the proof of the Sylvester-Gallai theorem:



It is at this stage that we have to elaborate on the "proof by picture" we previously gave.
We have allowed for the possibility that y coincides with `proint_lin_proj t`. We have devised two proof strategies, depending on whether this is the case or not. We will first discuss the case where the points coincide, as we do in our formal proof.

Informal intuition may lead us to consider using Pythagoras's theorem on what is in this case the triangle of points `t.1`, `proint_lin_proj t'` and `proint_lin_proj t = y`. It is here that we note that we must pay the price for not working in affine subspaces. Seeing as the only information we used about our lines was their convexity, we only have the following characterisation available, in which no orthogonality appears:

Characterisation of distance minimiser
`norm_eq_infi_iff_real_inner_le_zero`

The point $v \in K$ attains the minimum distance to u, if and only if it forms an obtuse angle with u and any other point w of K.
In code: ‖u - v‖ = (⊓ w : K, ‖u - w‖) ↔ ∀ w ∈ K, ⟨u - v, w - v⟩≤ 0

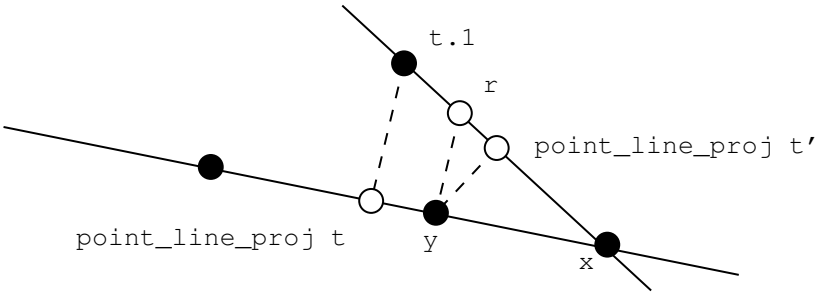
However, we have found a way to handle this constraint, in our lemma `SG_Pythagoras_workaround`, which uses:

Polarisation identity
`real_inner_eq_norm_mul_self_add_norm_mul_self_sub_norm_sub_mul_self_div_two`

In code: `⟨x, y⟩ = (‖x‖ * ‖x‖ + ‖y‖ * ‖y‖ - ‖x - y‖ * ‖x - y‖) / 2`

We now sketch the proof for this case, using notation s for `point_line_proj t'` and t for `t.1`.
Using the fact that $\langle y-s, t-s\rangle \leqslant 0$, obtained from the characterisation of minimisers applied to point y and line (t,x), and the polarisation identity applied for this dot-product, we find $\|y-s\|^2+\|t-s\|^2 \leqslant \|t-y\|^2$. Our goal is to show that $\|y-s\| < \|t-y\|$, as $\|t-y\|$ is the assumed minimum distance among point-line pairs fo the main proof, so that the latter identity will provide the desired contradiction. We may deduce this from $\|y-s\|^2+\|t-s\|^2 \leqslant \|t-y\|^2$ by showing that $t \neq s$.
The characterisation of minimisers provides $\langle y-s, x-s\rangle \leqslant 0$ (for point y and line (t,x)) and $\langle y-s, y-x\rangle \leqslant 0$ (for point `proint_lin_proj t = y` and line (s,x)), from which we deduce $\|y-s\|^2 \leqslant 0$, hence $y=s$. So if we had $t=s$, then $t=y$ and t would be on the line spanned by x and y, which contradicts the definition of the point-line pairs we consider the distances of. This concludes the first case of the final step of our proof.

Next, we discuss the case in which y and `point_line_proj t` do not coincide. We introduce the point r :



It is the intersection of the parallel through y to the line passing through `point_line_proj t` and `t.1`, and the line passing through x and `t.1`. In our lemma `SG_Thales_like`, using only some manipulations with vectors, we show that `‖r - y‖ < ‖t.1 - point_line_proj t‖`. In our proof of `Sylvester_Gallai`, we show that r is on the line spanned by `t.1` and x, so that `‖point_line_proj t' - y‖ ⩽ ‖r - y‖`, by minimality of `point_line_proj t'`. Piecing the inequalities together, we again contradict the minimality of triple t by having shown that triple t' represents a smaller point-line distance.
A formal subtlety in that part of the proof is the use of `cinfi_le`, which states `∀{f : ι → α}, bdd_below (set.range f) → ∀ (c : ι), infi f ≤ f c`. The lemma `infi_le` states the same, but requires the instance of a `complete_lattice` which is not the case for $\mathbb{R}$, so that we have to use the previous lemma, which requires only an instance of a `conditionally_complete_lattice`.

We can therefore conclude that in the proof of the main theorem, in both cases we may find a point-line pair with smaller distance then the one we assumed to be minimal among these pairs. We thus derive the desired contradictions, and the Sylvester-Gallai theorem is proven.

```
theorem Sylvester_Gallai
  (P : finset E) (hSG : ¬ (∃ a b : E, ∀ p ∈ P, p ∈ line a b)) :
  ∃ a b ∈ P, a ≠ b ∧ (∀ p ∈ P, p ≠ a → p ≠ b → p ∉ line a b) :=
```

As an anecdote, we mention that in our first proof, we forgot to mention condition `a ≠ b`. We only noticed this mistake when attempting to apply the theorem in a proof in file `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry`.

5.3 Bounding the size of 4-cycle free graphs

We refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_graphs`.

In this section we discuss the formalisation of content from section 5 of chapter 27 of "Proofs from THE BOOK", which illustrates applications of double counting to some problem of extremal graph theory. the first application of double counting mentioned in our source, the handshake lemma, is already implemented in mathlib as:

Handshake lemma (variant)
`simple_graph.sum_degrees_eq_twice_card_edges`

In a graphs (V, E), we have $\sum_{v \in V} \deg(v) = 2|E|$.
In code: `∀ (G : simple_graph V), ∑ v, G.degree v = 2 * G.edge_finset.card`

We shall need it soon, when formalising the following:

Reiman's theorem

For a graph (V, E) containing no cycles of length 4, we have $|E| \leqslant \left\lfloor \frac{|V|}{4} \left(1 + \sqrt{4|V| - 3}\right) \right\rfloor$.

Proof: We consider the following relation between vertices u and pairs of vertices $\{v, w\}$: u is a common neighbour to v and w. We shall perform double counting with this relation.
If we count the pairs in relation with u, we consider exactly the pairs of its neighbourhood, of which there are $\binom{\deg(u)}{2}$. Next, we will not count the common neighbours to a pair of vertices v and w, but upper-bound them. Indeed, for any pair of vertices v and w, if the graph has no 4-cycles, then there can be at most one common neighbour u: if there were at least one more, say u', then v, u, w, u' would form a 4-cycle.
Hence from double counting, we derive $\sum_{u \in V} \binom{\deg(u)}{2} \leqslant \binom{|V|}{2}$, as there are $\binom{|V|}{2}$ pairs of vertices.
We may rewrite this inequality to $\sum_{u \in V} \deg(u)^2 \leqslant |V|(|V| - 1) + \sum_{u \in V} \deg(u)$.
We may combine this with inequality $\left(\sum_{u \in V} \deg(u)\right)^2 \leqslant |V| \left(\sum_{u \in V} \deg(u)\right)$, which is derived from the Cauchy-Schwarz inequality applied $|V|$-dimensional vector $(\deg(u))_{u \in V}$ and the all-one vector $(1)_{u \in V}$,
so as to get $\left(\sum_{u \in V} \deg(u)\right)^2 \leqslant |V|^2(|V| - 1) + |V| \sum_{u \in V} \deg(u)$
Then, we use the handshake lemma to rewrite this to $4|E|^2 \leqslant |V|^2(|V| - 1) + 2|V||E|$.
This is a quadratic inequality in $|E|$, which we may solve to obtain the statement of the theorem. □

For our formalisation, we first define the meaning of a graph containing no 4-cycle:

```
def c4_free (G : simple_graph V) : Prop :=
  ∀ (v : V) (c : G.walk v v), ¬((c.is_cycle) ∧ (c.length = 4))
```

We drew inspiration from mathlib's `simple_graph.is_acyclic : ∀ (v : V) (c : G.walk v v), ¬c.is_cycle`.
In this definition, we avoid subgraphs. In mathlib, cycles are defined as walks from v to v, for some vertex v on the cycle. The above definition then says that all walks cannot be both cycles, and have length 4.

Our first lemma is `c4_free_common_neighbours`, which states that in a 4-cycle free graph, pairs of vertices have at most 1 common neighbour. The notion of common neighbours is already present in mathlib in the form of `common_neighbors`.

```
lemma c4_free_common_neighbours
  (G : simple_graph V) (h : c4_free G) :
  ∀ v w, v≠w → ((common_neighbors G v w).to_finset).card ≤ 1 :=
```

If the instance `tec_tec` and `tec`, and the `open_locale classical` are all commented-out, Lean will not recognise that there are finitely many common neighbours, and raise an error at `to_finset` in the statement of `c4_free_common_neighbours`. This error lead us to show `tec`. We encourage the reader to try out different combinations of out-commenting of instances and commands, and see what the effect on the theorem that follows: some of them are (perhaps) unexpected. The reasons for these behaviors are found within foundations, and we will not discuss them here. We will simply note that we have here an example of a derivation of a fintype instance, certifying finiteness, using mathlib support in the form of `fintype.of_surjective`, in our case.

Now, we discuss the formal details of `c4_free_common_neighbours`. The proof is by contradiction, and with a technical lemma `pair_of_two_le_card` of our making, we obtain two vertices a and b in the common neighbourhood of u and v. We use them to build our 4-cycle `c4`. Mathlib walks are defined inductively as:

```
inductive walk : V → V → Type u
| nil {u : V} : walk u u
| cons {u v w: V} (h : G.adj u v) (p : walk v w) : walk u w
```

So to build `c4`, we start with the one vertex walk `@walk.nil V G v` and "add edges" with `walk.cons`. As we can see in the claim we named `for_later_too`, the computational nature of the length of walks allows us to easily show that the walk has length 4.
Next, to show that this walk is a cycle, we use mathlib's `walk.is_cycle_def` to reduce this to showing that the walk has no duplicates among edges, is not a 1-vertex walk, and has no duplicates among vertices. The proofs, in which the fact that no duplicates exist is shown, have the same features, which we now comment on.
Metaprogramming makes its first appearance within a proof in this thesis, in the form of `repeat`, since we will repeatedly consider an element and prove that it differs from a list of other elements. Looping through the elements we which to compare is achieved with tactic `fin_cases`. To handle the different assumptions necessary to each different case, we use `<|>`: this will lead Lean to ignore the code on the left side of it, if it fails, and proceed with code on its right. Using this operation, we let Lean try all relevant assumptions to prove distinctness, in each `repeat` loop.
Another noteworthy aspect to this proof is the use of `sym2.eq_iff` to characterise equality of edges. Edges have their own type, `sym2`, of symmetric pairs of he vertex type.

Proceeding with the proof of Reiman's theorem, we will need double counting. This technique is implemented in mathlib in the form we now describe. Given two finite sets s and t, and a relation r between elements of s and t, we may count the pairs of element that are in relation in two ways. For each element from one of the sets, we may define the set of elements from the other set that it is in relation with:

```
/-- Elements of `s` which are "below" `b` according to relation `r`. -/
def bipartite_below : finset α := s.filter (λ a, r a b)

/-- Elements of `t` which are "above" `a` according to relation `r`. -/
def bipartite_above : finset β := t.filter (r a)
```

Summing the sizes of these sets over the respective index set yields the same sum, for both orders of s and t.

Double counting
`finset.sum_card_bipartite_above_eq_sum_card_bipartite_below`

We have $\sum_{x \in s} |\{y \in t \mid r\ x\ y\}| = \sum_{y \in t} |\{x \in s \mid r\ x\ y\}|$.
In code: `∑ (a : α) in s, (bipartite_above r t a).card = ∑ (b : β) in t, (bipartite_below r s b).card`

Back the proof of Reiman's theorem, we define our double counting relation on pairs of edges and vertices, stating that all (both) vertices of the pair e are adjacent to vertex u, in the following:

```
def double_counting_rel (G : simple_graph V ) (u : V) (e : sym2 V) :=  ∀ v ∈ e, G.adj u v
```

The next step in our formal proof will be to show that for a given u, there are $\binom{\deg(u)}{2}$ pairs in relation with it. Seeing as we did not require the elements of the symmetric pair to be different in the definition of our relation, we must restrict the double-counting to the pairs with non-equal elements, the negation of `sym2.is_diag`. We give two proofs of the exact same formal equivalent of this statement, in `double_count_above` and `double_count_above'`.

```
lemma double_count_above
  (G : simple_graph V ) (u : V) :
  ((finset.bipartite_above (double_counting_rel G) ({e : sym2 V | ¬ e.is_diag}.to_finset)) u ).card
  = (G.degree u).choose 2 :=
```

The first version will compute the size through the use of finsets (finite sets), whereas the second will compute it using the mathlib support for `sym2`. This illustrates that multiple approaches are possible, though the most efficient is that which best makes use of mathlib support.
In the first version, we put the pairs in relation with u in bijection with `finset.powerset_len 2 (G.neighbor_finset u)`, the set of length two subsets of the set of neighbours of u, using the operations `quotient.out` to extract a vertex of an edge of type `sym2`, and `mem.other` to obtain the other vertex of that edge.
In the second version, we compute the size by putting the pairs in relation with u in bijection with `image quotient.mk (G.neighbor_finset u).off_diag`, the symmetric pairs of elements of the set of neighbours of u with distinct pair-elements. For both cases, we take note of the lemmata that compute the size:

Size computations
`finset.card_powerset_len`
`sym2.card_image_off_diag`

In code, respectively:
`∀ {α : Type} (n : ℕ) (s : finset α), (powerset_len n s).card = s.card.choose n`
`∀ {α : Type} (s : finset α), (image quotient.mk s.off_diag).card = s.card.choose 2`

Note that `n.choose k` denotes the binomial coefficient $\binom{n}{k}$.

Moving on to the second part of double counting, which consists in bounding the number of vertices in relation with a given pair, we see in `double_count_below` that this step is merely a rephrasing of `c4_free_common_neighbours`.

```
lemma double_count_below
  (G : simple_graph V ) (hG : c4_free G) (v w : V) (vnw : v ≠ w ):
  ((finset.bipartite_below (double_counting_rel G) (univ)) ((v,w)) ).card ≤ 1 :=
```

Summing this bound over the pairs, we obtain the bound from `double_count_below_bound`, where we make use of more support for size computation in `sym2`, in the form of mathlib's `sym2.card_subtype_not_diag` : `∀ {α : Type}, fintype.card {a // ¬a.is_diag} = (fintype.card α).choose 2`.

We may then summarise the efforts made so far in `first_ineq`, which is the equivalent to the inequality $\sum_{u\in V}\binom{\deg(u)}{2} \leqslant \binom{|V|}{2}$ from our informal proof. This is where the formal lemma for double counting, `finset.sum_card_bipartite_above_eq_sum_card_bipartite_below`, is put to use.

From here on, the proof is purely computational and requires no more graph theory or combinatorics, both informally and formally. In `first_ineq_rw`, we perform the first rewrite of our inequality, which gets rid of the binomial coefficients. As take-away, we highlight:

Binomial coefficients
`nat.choose_two_right`

In code: `∀ (n : ℕ), n.choose 2 = n * (n - 1) / 2`

Note that this is not a fraction: since $n(n-1)$ is even, quotient and fraction coincide. Having small lemmata such as this one in mathlib greatly alleviates the formalisation process.

Next, we will make use of the Cauchy-Schwartz inequality. Mathlib has an implementation of this inequality in the form of `norm_inner_le_norm`, which may be found in `analysis.inner_product_space.basic`. However, this version requires an inner product whose output field must have instance `is_R_or_C` (meaning that it is algebraically similar to $\mathbb{R}$ or $\mathbb{C}$), which is not the case for $\mathbb{N}$ or $\mathbb{Z}$.

At this stage, we could have coerced our inequality to $\mathbb{R}$, so as to make use of `norm_inner_le_norm`. However, seeing as we knew a different proof for the Cauchy-Schwartz inequality, which worked for integers, and we know that the Cauchy-Schwartz inequality is used rather frequently in combinatorics, where terms are usually naturals or integers, we decided to prove our own the Cauchy-Schwartz inequality. This is also somewhat coherent with the mathlib philosophy of stating facts as general as possible. Thus we show:

Cauchy-Schwartz inequality

For integers $a_1, ..., a_n$ and $b_1, ..., b_n$ we have $\left(\sum_{i=1}^{n} a_i b_i\right)^2 \leqslant \left(\sum_{i=1}^{n} a_i^2\right)\left(\sum_{i=1}^{n} b_i^2\right)$.

Proof: We develop the sum of squares $\sum_{i=1}^{n}\sum_{j=1}^{n}(a_i b_j - a_j b_i)^2 = \sum_{i=1}^{n} a_i^2 \sum_{j=1}^{n} b_j^2 + \sum_{j=1}^{n} a_j^2 \sum_{i=1}^{n} b_i^2 - 2\sum_{i=1}^{n} a_i b_i \sum_{j=1}^{n} a_j b_j$. The latter simplifies to $2\left(\sum_{i=1}^{n} a_i^2\right)\left(\sum_{i=1}^{n} b_i^2\right) - 2\left(\sum_{i=1}^{n} a_i b_i\right)^2$. Since the sum of square was positive, we get the desired inequality. □

We have formalised this as `Cauchy_Schwartz_int`. We will not discuss its proof, mentioning only that it serves as good example for how to manipulate sums. Instead, we will put the inequality to use in our context, in the form of `Cauchy_Schwartz_in_action`, where we obtain the $\left(\sum_{u\in V}\deg(u)\right)^2 \leqslant |V|\left(\sum_{u\in V}\deg(u)\right)$ from the informal proof.

Moving on, `second_ineq` combines the previous inequalities and `third_ineq` rewirtes the inequality using the handshake lemma, `sum_degrees_eq_twice_card_edges`, so as to get the number of edges to appear. It is also in `third_ineq` that we finally coerce to $\mathbb{R}$, which will be necessary in order to take square roots.

From here on, the last step is purely algebraic rewriting, so that we may state it without the actual numbers of interest for the theorem, in `max_edges_of_c4_free_Istvan_Reiman_pre`.

```
lemma max_edges_of_c4_free_Istvan_Reiman_pre
  (a b : ℕ)
  (ineq :   (4*(a)^2 : ℝ) ≤ ((b)^2)*((b) - 1) + (b)*2*(a)):
  ((a) : ℤ) ≤ ⌊((b) / 4 : ℝ)*(1 + real.sqrt (4*(b) - 3))⌋ :=
```

This will ease notation greatly, an aspect particularly relevant to formalisation. The actual formal statement of Reiman's theorem, in `max_edges_of_c4_free_Istvan_Reiman`, will then follow quickly from the previous work.

It is now time to dig out high-school level math. We are given an inequality of form $4a^2 \leqslant b^2(b-1) + 2ba$, and our goal is to derive $a \leqslant \left\lfloor \frac{b}{4}\left(1+\sqrt{4b-3}\right)\right\rfloor$. To do so, we bring our inequality in canonic form $4\left(a-\frac{b}{4}\right)^2 \leqslant \frac{b^2}{4}(4b-3)$. Taking *mathlib's* square root, rearranging, and considering integrality, provides the desired result.
Indeed, mathlib's square root is defined for negative numbers, where it has value 0. This has effects such as:

Properties of mathlib's square root
`real.le_sqrt_of_sq_le`
`real.sqrt_mul`

In code, respectively:
```
∀ {x y :  ℝ}, x ^ 2 ≤ y → x ≤ real.sqrt y
∀ {x :  ℝ}, 0 ≤ x → ∀ (y :  ℝ), real.sqrt (x * y) = real.sqrt x * real.sqrt y
```

We use these in our formal proof of `max_edges_of_c4_free_Istvan_Reiman_pre`. We also mention the use of `int.le_floor` to transition from the floor function to coercion. Finally, we may conclude with:

```
theorem max_edges_of_c4_free_Istvan_Reiman
  (G : simple_graph V ) (hG : c4_free G) :
  ((G.edge_finset.card) : ℤ) ≤ ⌊((fintype.card V) / 4 : ℝ)*(1 + real.sqrt (4*(fintype.card V) - 3))⌋
```

Next, we discuss whether the bound from Reiman's theorem is tight.
For this purpose, we will construct the following family of graph, indexed by primes p: we let G_p be a graphs with vertices being the points of the projective plane over $\mathbb{Z}/p\mathbb{Z}$, and let two of them be adjacent iff they lie on the polar line of one another.
As a reminder, the points of the projective plane over $\mathbb{Z}/p\mathbb{Z}$ are obtained as the span of the non-zero vectors of $(\mathbb{Z}/p\mathbb{Z})^3$. Then, $span(w)$ and $span(v)$ are adjacent in the graph iff they are distinct and $\langle w, v\rangle = 0$, where this condition is independent of the representative of the spans.

We shall only prove the following, which is merely a step in the corresponding chapter of "Proofs from THE BOOK", which we could not treat fully due to a lack of time. Its use of linear algebra makes it a useful example, however, which is why we decide to include it in this thesis.

Extremal graphs

The graphs G_p we described are 4-cycle free.

Proof: Equivalently, we will show that any two vertices of the graphs can have at most one common neighbour. A common neighbour $span(w)$ to distinct vertices $span(u)$ and $span(v)$ must satisfy the system

$$\begin{cases} w_1v_1 + w_2v_2 + w_3v_3 = 0 \\ w_1u_1 + w_2u_2 + w_3u_3 = 0 \end{cases}$$

The fact that $span(u) \neq span(v)$ implies that u and v are linearly independent, so that the latter system has a 1-dimensional solution space.
Therefore, $span(w)$ is the *only* common neighbour of distinct vertices $span(u)$ and $span(v)$. □

We start our formalisation with `common_neighbors_c4_free`, in which we show that if in a graph, any two vertices of the graphs have at most one common neighbour, then the graph is 4-cycle free. We prove this by contraposition, so that we assume the graph to have a 4-cycle, and prove that there exists a pair of vertices with more then 1 common neighbour.
We call the cycle `cyc` and, due to its inductive nature, unfold it with `cases`, which has the side effect of requiring us to certify that for each extracted vertex, the corresponding walk is not a 1-vertex walk. To prove that the extracted vertices v and d are the same, so that the walk represents a cycle, we use the length of the walk, that we kept track of in our hypotheses, and mathlib lemma `simple_graph.walk.eq_of_length_eq_zero : ∀ {p : G.walk u v}, p.length = 0 → u = v`. We then use the vertices v and b that are opposite of one another on the 4-cycle as candidates for the pair of vertices with more then 1 common neighbour. Next, we show that the two other vertices a and c of the 4-cycle are among the common neighbours of v and b, so that the set of common neighbours has size at least 2.

Next, we enter the development of finite geometry. The type of points of the projective plane is `projectivization (zmod p) (fin 3 → (zmod p))`, where `zmod p` denotes $\mathbb{Z}/p\mathbb{Z}$ and `(fin 3 → (zmod p))` denotes $(\mathbb{Z}/p\mathbb{Z})^3$. The latter is inspired by mathlib's stadard way of defining vectors. Then, we define the adjacency relation for our graph:

```
def edge_relation  (v w : (projectivization (zmod p) (fin 3 → (zmod p)))) :=
  (v ≠ w) ∧ (matrix.dot_product v.rep w.rep = 0)
```

We may use it to define the `extremal_graph`. Indeed, mathlib graphs are defined from an adjacency relation over the type of vertices. When defining a graph, one has to provide a proof that the relation use is symmetric, in field `symm`, and irreflexive, in field `loopless`. In our case, symmetry follows from `matrix.dot_product_comm`, and irreflexivity from the relation directly. We then create a rewrite-lemma in the form of `neighbor_extremal_graph`, which characterises adjacency in our graphs, to ease exposition.

Seeing as we will speak of the size the set of common neighbours, our characterisation `common_neighbors_c4_free` required the vertex type to be a `fintype`. We now need to prove that our projective plane is a finite type in instance `Zp3_fin`. There, we make use of mathlib's `quotient.fintype`. We remark that Lean could not infer the setoid of the quotient, which is why we give it explicitly.

Now, we will make use of linear algebra. We remind Lean that we use the dot-product as bilinear form in our definition `dotp`. In a moment, in our lemmata `dotp_is_refl` and `dotp_nondegenerate`, we will show that it is reflexive and non-degenerate respectively.

A key step of the informal proof is that the solution space of the mentioned system, which corresponds to the orthogonal complement to the span of two linearly independent vectors, is 1-dimensional. We formalise this in our `dim_of_ortho`. Here, we make use of `dotp_is_refl` and `dotp_nondegenerate` as well as the mathlib support:

Dimension
`bilin_form.finrank_add_finrank_orthogonal`
`finite_dimensional.finrank_fin_fun`
`finrank_span_eq_card`

We have, respectively:

- With respect to a reflexive bilinear form B, for a subspace W of V, we have $\dim(W) + \dim\left(W^{\perp}\right) = \dim(V) + \dim\left(W \cap V^{\perp}\right)$.
- For field K, the dimension of K^n is n.
- For a finite family $(b_i)_{i \in I}$ of linearly independent vectors, we have $\dim(span((b_i)_{i \in I})) = |I|$.

In code, respectively:

- `{B : bilin_form K V} {W : subspace K V}, B.is_refl → finite_dimensional.finrank K W + finite_dimensional.finrank K (B.orthogonal W) = finite_dimensional.finrank K V + finite_dimensional.finrank K (W ⊓ B.orthogonal ⊤)`
- `finite_dimensional.finrank K (fin n → K) = n`
- `{b : ι → V}, linear_independent K b → finite_dimensional.finrank K (submodule.span K (set.range b)) = fintype.card ι`

Next, the fact that in our context, representative vectors are linearly independent iff the corresponding vertices are distinct, is formalised as our `rw_tec`. The transition between independence in the projective plane and independence of representatives is achieved through `projectivization.independent_iff`, and the transition between independence of a pair of point of the projective plain and their distincness is achieved with `projectivization.independent_pair_iff_neq`.

Then, we need one last technical step, in the form of our `ortho_span_pair_iff`. There, we show, in our context, that a vector is in the orthogonal complement of the span of a pair of vectors iff it is orthogonal to the two vectors of that pair.

Finally, we may piece our lemmata together to prove that our extremal graphs G_p are 4-cycle free.
We show this in our `extremal_graph_c4_free`. In its proof, we use `common_neighbors_c4_free` to reduce the task to showing that for any two vertices v and w, there is at most one common neighbour. We assume that there are two or more for contradiction, naming these two a and b. Then, considering representatives, we show that a and b are in the orthogonal complement of the span of v and w. Recalling the dimension of that complement in our claim `dim_o`, we use the following characterisation to conclude that the representatives of a and b are in fact linearly dependent, which contradicts the assumption that a and b were different.

Characterisation of 1-dimensional spaces
`finrank_eq_one_iff_of_nonzero'`

Space V is 1-dimensional iff for any $v \neq 0$ of it (and one such vector exists), all vectors $w \in V$ may be writen as $w = c \cdot v$ for some scalar c.
In code:
`(v : V), v ≠ 0 → (finite_dimensional.finrank K V = 1 ↔ ∀ (w : V), ∃ (c : K), c • v = w)`

This concludes our formalisation effort for the corresponding section of "Proofs from THE BOOK". The next step would be to compute the number of vertices of graph G_p. This requires computing the `fintype.card` of a quotient type. The only support for sizes of quotient types we could find in mathlib are `fintype.card_quotient_le` and `fintype.card_quotient_lt`, which are inequalities. The informal computation states that $(\mathbb{Z}/p\mathbb{Z})^3$ has $p^3 - 1$ non-zero vectors, and each 1-dimensional span contains $p - 1$ vectors, so that G_p has $\frac{p^3-1}{p-1}$ vertices. One would first have to write support for the size of quotient types in which each equivalence class has equal size, a time consuming endeavour due to it requiring a certain familiarity with quotient types and fintypes.

6 Conclusions

Lean and formalisation
We found that formalising mathematics in Lean is a simple but time consuming task.
We note that a large amount of time had to be dedicated to learning the basics of Lean, the possible problems and their solutions one may run into when using it, and the mathematics formalised in mathlib. However, we believe that this learning phase will get shorter over time, as the amount of documentation is bound to increase.
At the beginning of our work, we recall having considered formalisation to be a quite exhausting task. We believe this to be a common experience shared by many. The degree of explicity and the lack of some form of automation for certain tedious routine tasks make formalisation a rather frustrating experience, as one has to spend much time on steps one considers almost obvious. However, we found that one gets used to this aspect quite quickly. Once the learning phase is completed, formalisation becomes much less time consuming and tiresome.

We remark that the Lean community is rapidly evolving. Influential figures such as Terrence Tao have started formalising mathematics in Lean [1], which will almost certainly increase the number of users in the months to come. We hope that the Lean community is ready to scale up, faced with this increased attention. We would be happy to contribute to this community regularly. Indeed, we have found writing API (mathlib support) to be a task whose importance is not to be underestimated, and is beneficial to the community as a whole. We would also like to contribute by producing pedagogical resources for Lean and mathlib.

To close this discussion, we mention that the development automation, such as Leandojo [5] and Aesop [6], may change the formalisation process drastically in the years to come. These tools currently successfully allow to prove small lemmata fully automatically. In a future that is fairly reasonable to envision, formalisation will not require proof-writing anymore, but merely the process of formalising the statements of sufficiently many intermediate steps in a target proof.

Formalising "Proofs from THE BOOK"
We found the formalisation of proofs from "Proofs from THE BOOK" to be an enlightening experience. It pushed us to gain familiarity with a wide range of areas of mathlib.
One may consider that in some sense, a certain aspect of proofs from "Proofs from THE BOOK" was ideal for formalisation. In the latter one seeks relatively short and elementary proofs, or alternatively, proofs based on powerful tools, possibly from different mathematical areas, as these factors ease the formalisation. One may consider these aspects to coincide with a criterion of mathematical beauty that is implicit to "Proofs from THE BOOK".
Alternatively, one may draw a different conclusion: formal mathematics is not aesthetic. In the context where "Proofs from THE BOOK" highlights the beauty of mathematical ideas, intuition and insight, formalisation may be considered as a process with an entirely different goal, which is that of certifying correctness.

We must however draw the subjective conclusion that Lean and mathlib are currently at a stage not sufficiently developed to tackle much of content from "Proofs from THE BOOK" with ease, or in an idiomatic way. The latter is possible for for certain content, such as the content on the pigeonhole principle and double counting from chapter 27 of "Proofs from THE BOOK", but is not the norm, as can be seen from our section 5. This is not an issue if one does not care for the production of idiomatic code, say if one merely wishes to certify correctness, but it is problematic in the light of using content of "Proofs from THE BOOK" for pedagogical purposes.

Research and development opportunities
The most opportunities can be found in formalisation itself. Mathlib still does not contain much undergraduate mathematics, and its community is looking for users to formalise it. Opportunities for formalisation projects on advanced or recent mathematics are seemingly endless. We refer to [7] for a list of large formalisation projects, and to [2] for a list of journal publications about formalisation projects.

There are many projects surrounding Lean that we found hard to classify. We will give some examples:

- **Easing interaction with Lean**
  Projects such as *Paperproof* [8] aim to visually represent formal proofs in Lean, with the formal mathematics equivalent of a flow-chart.
  The *Lean verbose* project [9] experiments with the translation of informal verbose statements into formal Lean code.

- **Programming in Lean**
  As we mentioned in our introduction, Lean may be used as a functional programming language. In projects such as [10], the game of Sudoku has been implemented in Lean. More generally, the Lean community is pushing for Lean to be used as a programming language, as can be seen from texts such as [11].
  A mathematically relevant project in this area is SciLean [12], which aims at implementing algorithms from scientific computing (in german, Numerik) in Lean. We recommend visiting the repository [12] which sketches how a harmonic oscillator simulation is implemented in Lean. Another example in this field is the following implementation of Gaussian elimination in [13]. In [15], a SAT solver is implemented and verified.
  We believe it to be only a matter of time until attempts at implementing (and formally verifying) algorithms from computer algebra and discrete and numerical optimisation will be made.
- **Algorithmic automation for Lean**
  As a last example of projects, we mention [3] whose aim is to allow Lean to use Mathematica as a black box. Mathematica contains many algorithms from computer algebra and optimisation, some of which may be used as black boxes to produce certificates that ease proof-writing. As a toy example to illustrate this concept, consider the task of finding the prime decomposition of a number in Lean. We can use Mathematica to compute this prime decomposition, and then simply compute its product (and certify primality of factors) in Lean to certify that it truly is a decomposition of the given number.

To close the discussion on research opportunities, we will mention the AI community's interest in Lean. Indeed, in the light of considering formalising mathematics in Lean to be a game, as is evidenced in the *natural numbers game* [14], it is natural that an AI community previously focused on games such as chess and go will gain increasing interest in formal mathematics. Publications on the subject can be found at [4], and we recall the running example on the matter, Leandojo [5].
Another aspect to AI research on Lean is the translation between fromal and infromal mathematics. For example *lean-chat* (repository at [16] and blog post at [17]) allows users to translate an informal statement into a formal one.

We find the Idea of an AI producing relevant and correct mathematics to be fascinating, and would very much like to work in this field. We believe this field to be full of new challenges, as the search spaces one works with are much larger, and have the interesting feature of being searched though purely algorithmically in certain parts (such as proof of inequalities with tactic `linarith`).

References:

1 https://github.com/teorth/symmetric_project

2 https://leanprover-community.github.io/papers.html

3 https://robertylewis.com/leanmm/

4 https://paperswithcode.com/task/automated-theorem-proving

5 https://leandojo.org/

6 https://github.com/leanprover-community/aesop

7 https://leanprover-community.github.io/lean_projects.html

8 https://github.com/Paper-Proof/paperproof

9 https://github.com/PatrickMassot/lean-verbose

10 https://github.com/TwoFx/sudoku

11 https://lean-lang.org/functional_programming_in_lean/

12 https://github.com/lecopivo/SciLean

13 https://github.com/jjcrawford/lean-gaussian-elimination

14 https://adam.math.hhu.de/#/g/hhu-adam/NNG4

15 https://github.com/siddhartha-gadgil/Saturn

16 https://github.com/zhangir-azerbayev/lean-chat

17 https://xenaproject.wordpress.com/2022/08/16/the-future-of-interactive-theorem-proving/