Formal proofs from THE BOOK: Appendix

Yves Jäckle

This document contains discussions for each formalisation of content from "Proofs from THE BOOK". It is not officially part of the master thesis, but may still be of use, which is why we decided to make it public.

Contents

1 Chapter 1: Six proofs of the infinity of primes

The goal, throughout the section, is to give different proofs to the fact:

Infintude of primes:

There are infinitely many prime numbers.

We've formalized the first 5 proofs out of the 6 proofs of the infinitude of primes from the corresponding chapter in "Proofs from THE BOOK". As the title suggests, the goal is to show that there are infinitely many prime natural numbers, each time with a different proof using different arguments, making use of different areas of math.

1.1 Euclid's proof

Refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_1stProof`.

Let us recall the proof.
Proof: Assume, for contradiction, that there exist only finitely many prime numbers $p_1, ..., p_n$.
We can consider their lcm $\prod_{i=1}^{n} p_i$, and its successor $\left(\prod_{i=1}^{n} p_i\right) + 1$.
On the one hand, any number has a prime divisor, so that there is some prime p_i that divides $\left(\prod_{i=1}^{n} p_i\right) + 1$.
This prime also divides the product $\prod_{i=1}^{n} p_i$, as it's one of its factors.
Yet, since a number and its successor are coprime (hence, have as only common divisor 1), the prime p_i would equal 1, which isn't prime ! We've reached the desired contradiction. □

This proof serves as basis for mathlib's version of infinitude of primes, `nat.exists_infinite_primes`.

Infinitude of primes
`nat.exists_infinite_primes`

For each number n, we may find a prime number p greater (or equal) to n.
In code: `∀ (n : ℕ), ∃ (p : ℕ), n ≤ p ∧ nat.prime p`

We first prove that for any finset s, there is a prime p not contained in s, in `Euclid_proof`, from which we derive the fact that the set $\{n \mid n\ prime\}$ is infinite, in `Euclid_proof_standardised`.

The proof of `Euclid_proof` is by contradiction. Now, assuming that s is a finset containing all primes, a hypothesis we name `h`, we filter out the composite (:= non-prime) numbers in s by defining `s_primes`.
We show that `s_primes` contains all primes, and only primes, in lemma `mem_s_primes`.

Next, we could introduce `(∏ i in s_primes, i) + 1` as a named constant, but we felt like this would cause only unnecessary rewrites, so we work with it directly. We want to get a prime divisor of this expression using `nat.exists_prime_and_dvd`, which requires us to show that the expression isn't 1.

Existence of a prime divisor
`nat.exists_prime_and_dvd`

All numbers but 1 have a prime divisor.
In code : `∀ {n : ℕ}, n ≠ 1 → (∃ (p : ℕ), nat.prime p ∧ p ∣ n)`

We show this requirement in a lemma named `condition`.
This lemma is also shown by contradiction, which leads us to assume `(∏ i in s_primes, i) = 0`, which will clash with the lemma `however` that states `0 < (∏ i in s_primes, i)`. The latter is show with the fact that primes are positive, `nat.prime.pos`, and the fact that products of positives are positive, `finset.prod_pos`. The contradiction is obtained through irreflexivity of $<$, `lt_irrefl`.

Now, we use `nat.exists_prime_and_dvd` to obtain a prime divisor of the expression, that we name p. Since, p is a member of the product of primes, it divides the product, as stated in `dvd_prod_of_mem`, a fact we record as a nameless lemma.

The contradiction we seek will be derived from `problem`, which state that p divides 1.
This is because of `nat.dvd_sub' : ∀ {k m n : ℕ}, k ∣ m → k ∣ n → k ∣ m - n`, where m is the successor of the product of primes, and n is the product of primes, so that $m - n = 1$.
Finally we may quickly derive the contradiciton from `problem`, as mathlib has:
`nat.prime.not_dvd_one : ∀ {p : ℕ}, prime p → ¬p ∣ 1`.

Now, we may show `Euclid_proof_standardised`. The transition to the standard form is achieved by reasoning by contradiction, using the previous theorem on the `set.finite.to_finset` version of the assumed finite set, to find a prime not included in the set of all primes (the desired contradiciton).

1.2 2nd proof

Refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_2ndProof`.

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

Here, we show that we can produce an infinite sequence of pairwise different primes in `second_proof`, which we then translate to the standard version of infinitude of primes in `second_proof_standardised`.

We define the Fermat numbers `F` in their closed form $F_n = 2^{2^n} + 1$. Before moving to the actual proof, we show some technical facts that will come in useful later on. First, there's the bound $F_n > 2$, which will come in handy when dealing with subtraction on naturals, in the form of `fermat_bound`, and then there's the fact that Fermat numbers are odd, in the form of `fermat_odd`, which is used for showing that Fermat numbers are pairwise coprime.

We prove the recursive relation `∏ k in range n, F k = F n - 2` satisfied by Fermat primes in `fermat_product`. This is shown by induction. The base case can be handled by the tactic `trivial`.
In the inductive step, we extract the last term of the product with `prod_range_succ`, and rewrite the inductive hypothesis. We then show the desired conclusion in a form that makes no use of subtraction, in the form of lemma `h` stating `(F n)*(F n) + 2 = (F n.succ) + 2 * (F n)`, which is shown with the closed form of Fermat numbers and some algebraic manipulations.
Finally, to transition to the desired form, we let `linarith` do the heavy lifting, reminding it that it may handle the subtractions on naturals with `h_natsub_1` and `h_natsub_2`.

We may now show that different Fermat numbers are coprime, in `fermat_coprimes`.
The constraint `(h : k < n)` will be handled by the `wlog` tactic in the main proof.
We consider the greatest common divisor m of the two Fermat numbers and show lemma `h_m` stating that $m|2$. Indeed, when writing the larger Fermat number in its recursive form, we see that the gcd m divides the product of Fermat numbers the recursive form contains, for similar reasons as in Euclid's proof of infinitude of primes. We may then use the same idea as in Euclid's proof to conclude that $m|2$.
This means that $m = 1$ or $m = 2$, which we derive from the composite theorem `dvd_prime prime_two`, which makes use of 2's primality. Finally, to see that the second case can't occur, we make use of the fact that Fermat numbers are odd, the `fermat_odd` we proved, and yet in this supposed case, they should also be even as they have 2 as divisor, using `even_iff_two_dvd`, and a number can't be both even and odd, which we have in the form of `even_iff_not_odd`.

We've arrived at the first infinitude proof, `second_proof` in which we claim that there exists a sequence of pairwise different prime numbers. Since for all n, we may consider a prime divisor of Fermat number F_n, the axiom of choice `classical.axiom_of_choice` provides us with a sequence of such prime divisors.

Axiom of choice
`classical.axiom_of_choice`

If we know of the property that for all x of type T, there exists a y of type T', such that x and y are in relation r, then we may conclude with the existence of a function from T to T' mapping an x to $f(x)$ (of type T'), so that x and $f(x)$ are in relation r.
In code :
`(∀ (x : α), ∃ (y : β x), r x y) → (∃ (f : ∏ (x : α), β x), ∀ (x : α), r x (f x))`

The only real justification left to provide that this is the sequence we're looking for, is that these prime divisors are all pairwise different. They are, since they're prime divisors of coprime numbers.
So, with `wlog` allowing us to order two Fermat numbers $F_k < F_q$, we assum for contradiction that the prime divisors are equal, which we keep as assumption `problem`, and make use of `nat.eq_one_of_dvd_coprimes` to arrive at the stage where we have a prime number dividing 1, which is the contradiciton we desire.

Common divisors of comprime numbers
`nat.eq_one_of_dvd_coprimes`

A common divisor of comprime numbers is necessarily equal to 1.
In code: ∀ {a b k : ℕ}, a.coprime b → k | a → k | b → k = 1

Finally, we bring the infinitude proof in standard form in `second_proof_standardised`.
Here, we take one such a sequence we showed existent of in `second_proof`, naming it f, and use it as the injection required in `set.infinite_of_injective_forall_mem` , which states that a set is infinite if we may inject an infinite type (here ℕ) in it.

1.3 3rd proof

Refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_3rdProof`.

Let us recall the proof.
Proof: We again assume for contradiction that there are finitely many primes, which allows us to consider the largest among them, which we will denote by p.
Next, we consider the Mersenne number $2^p - 1$, and a prime divisor q of it.
We may then write $2^p \equiv 1 \pmod{q}$. In terms of the group $((\mathbb{Z}/q\mathbb{Z})\backslash\{0\}, \times)$, the latter rephrases to the order of 2 dividing p. Since p is prime, the order of 2 is either 1 or p.
If it were 1, then $2 \equiv 1 \pmod{q}$, so that $1 \equiv 0 \pmod{q}$ which only happens for $q = 1$, a case excluded by q's primality. So the order of 2 is p. Now, we recall from group theory that the order of an element divides the size of the group. Therefore, $p|q - 1$, from which we deduce $p \leqslant q - 1$ and then $p < q$.
This last inequality contradicts the maximality of p, as q is also prime. □

Just as for the first (Euclid's) proof, we first show that for all finite sets, we main find a prime it doesn't contain, in `third_proof`, and bring it in standard form, stated as `third_proof_standardised`.

In order to consider the supposed largest prime, we first filter the supposed finite set containing all primes, so that it contains only primes, naming it `s_primes`. Then, we show that this set is nonempty in a claim named `s_primes_nonempty`, a fact we require to define its largest element in the first place. Seeing as mathlib provides us with the fact that 2 is primes through `nat.prime_two`, we display it as a member of `s_primes`.

We then make use of `finset.max'` to define the largest element of `s_primes`. Watch for the `'` (prime): `finset.max` also exists, and does not require the set to be non-empty. For empty sets, `finset.max` returns the canonic lower bound $\bot$ of the type (provided the type has one). For example, for the type $\mathbb{N}$, $\bot = 0$.
We then note that p is prime in `p_prime`, who's proof makes use of `finset.max'_mem`, which recalls that the maximum is part of the set it is the maximum of.

To obtain the prime divisor q of Mersenne number $2^p - 1$, which is the goal of claim `dvd_Mers`, we again make use of `nat.exists_prime_and_dvd`. This requires us to prove that $2^p - 1 \neq 1$, which we show by contradiction. We assume $2^p - 1 = 1$, from which we deduce $2^p = 2$, using the naturals-specific rewrite `nat.sub_eq_iff_eq_add`, where we use mathlib's generously provided `nat.one_le_two_pow`, which states $1 \leqslant 2^p$.
Now, to deduce $p = 1$ from $2^p = 2$, so that we reach the desired contradiction (since 1 isn't prime) we use:

Injectivity of powers
`nat.pow_right_injective`

For $x \geqslant 2$, if $x^a = x^b$, then $a = b$.
In code: `∀ {x : ℕ}, 2 ≤ x → function.injective (λ (n : ℕ), x ^ n)`

We may now transition to the group $((\mathbb{Z}/q\mathbb{Z})\backslash\{0\}, \times)$, which has its own type denotes `zmod q` in Lean, trough: `zmod.nat_coe_zmod_eq_zero_iff_dvd : ∀ (a b : ℕ), ↑a = 0 ↔ b ∣ a`. We may now make use of group theory, and investigate the order of 2, which is `order_of (2 : zmod q)`.

We start by showing that it divides $q - 1$, the size of the group, in the claim named `odq`. We have:

Fermat's little theorem (variant)
`zmod.order_of_dvd_card_sub_one`

For a non-zero element of $\mathbb{Z}/p\mathbb{Z}$, where p is prime, the order divides the group size of $p - 1$.
In code: `∀ {p : ℕ} [_inst_1 : fact (prime p)] {a : zmod p}, a ≠ 0 → order_of a ∣ p - 1`

Thus, we have to show that we do not have $2 \equiv 0 \pmod{q}$ in our context. This could be possible if $q = 2$, which doesn't contradict its primality, but will contradict the parity of the Mersenne number $2^p - 1$, that q divides.
This argument is the content of claim `q_tec`. A noteworthy passage of its proof is the use of `zmod.eq_zero_iff_even`, which translates between parity and values modulo 2.

Deducing that we do not have $2 \equiv 0 \pmod{q}$ from $q \neq 2$ could require an entire formal proof to itself.
Fortunately for us, we are working in a ring of characteristic q, a fact recorded as `zmod.ring_char_zmod_n`, so that we may use `ring.two_ne_zero : ring_char R ≠ 2 → 2 ≠ 0`.

We may now provide the second key argument of the main proof: the order of 2 divides p. We get it from:

Order divides power
`order_of_dvd_of_pow_eq_one`

If in a multiplicative group G with neutral element 1, $x^n = 1$, then the order of x divides n.
In code : `x ^ n = 1 → order_of x | n`

With this fact, we deduce that the order of 2 can either be 1 or 2, using `nat.dvd_prime`.
To discard the first case, we make use of the

Characterisation of order
`order_of_eq_iff`

The order of an element x is the number n such that $x^n = 1$ and for all $0 < m < n$, we have $x^m \neq 1$.
In code : `0 < n → (order_of x = n ↔ x ^ n = 1 ∧ ∀ (m : ℕ), m < n → 0 < m → x ^ m ≠ 1)`

We use it to show that if the order of 2 were 1, then $2 = 1$, from which the tactic `norm_num` derives the contradiction.

Now that we know that the order is p, we may use $p|q-1$ to get $p \leqslant q-1$, which is achieved with `nat.le_of_dvd`. Due to the nature of truncated subtraction for naturals, the passage to $p < q$ is a little less smooth, and we have to remind Lean that primes are greater then 1, in the form of `prime.one_lt`.

Since p was the largest prime, we also have $q \leqslant p$, as q is prime. We derive this through the use of the technical lemma `finset.le_max'`. Finally, we may derive the contradiction for the main proof with the previous facts and `not_le_of_lt : a < b → ¬b ≤ a`.
Since `third_proof`'s statement is the same as the formalisation of that of Euclid's proof, deriving the standardised version `third_proof_standardised` is done exactly as in the first proof.

1.4 4th proof

Refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_4thProof`.

Let us recall the proof.
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 them, 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 can't be finitely many primes, as otherwise, their number would bound the harmonic series. □

This is one of the more formalisation heavy proofs of this thesis. We've had to modify the informal proof in some parts, and develop our own formal tools along the way. The main result is theorem `fourth_proof`, in which we show the standardised version of infinity directly. Due to the size of the formal proof (1300+ lines), we will discuss mostly its structure and the formalisation challenges we encountered, not the Lean details.

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 isn't 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 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 doen't exceed the length of the list. The latter is the reason we require hypothesis `h` as input to our function.

Th informal proof also requires ordering a product: $\prod_{p \in P_n} \frac{p}{p-1} = \prod_{k=1}^{\pi(n)} \frac{p_k}{p_k - 1}$.

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: and equivalent for finsets 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 an 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` of the file. 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 $\displaystyle\prod_{p\in P_n} \frac{p}{p-1} = \prod_{k=1}^{\pi(n)} \frac{p_k}{p_k-1}$ in lemma `order_the_prod` of the file.
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 couldn't find in mathlib, and named `list_stuff_2` and `list_stuff_3`.

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

We then reduce the telescopic product. We could have stated the reduction as an equality, but we performed it in the previous inequality, for purely arbitrary reasons (formalisation fever ?).
The reduction has `prod_ordered_primes_bound` as result: $\displaystyle\prod_{k=1}^{\pi(n)} \frac{p_k}{p_k-1} \leqslant \pi(n)+1$
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`. Next, we had to make use of our `prod_fin_range` to handle its mathlib equivalent `fin.prod_univ__prod_range` bizarre failure.

Informally speaking, we've derived the inequality $\displaystyle\prod_{p\in P_n} \frac{p}{p-1} \leqslant \pi(n)+1$.
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 the product is $\displaystyle\prod_{p\in P_n} \left(\sum_{i=0}^{\infty} \frac{1}{p^i}\right)$.
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 couldn't 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 nned 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 interediate 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`.
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. `nat.factorization_prod_pow_eq_self` 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 the familiar `nat.exists_prime_and_dvd` in the induction step.

Finally, we may carry out our proof strategy, in the proof of `the_great_split_part_3`.
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 $\sum_{i\in[2^{n+1}]} \frac{1}{i} = \sum_{i\in[2^n]} \frac{1}{i} + \sum_{i\in[2^n+1,\ 2^{n+1}]} \frac{1}{i} \geqslant \frac{n}{2} + \sum_{i\in[2^n+1,\ 2^{n+1}]} \frac{1}{2^{n+1}} = \frac{n+1}{2}$. □

We may finally conclude with the infinitude of primes in `fourth_proof`.
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 on of our longer formalisations of the thesis.
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's 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.

1.5 5th proof

Refer to the file named `FormalBook_Ch1_InfinitudeOfPrimes_5thProof`.

Let us recall the proof.
Proof: The proof makes use of point-set-topology!
We will work with the two-way infinite arithmetic progressions we denote $N_{a,b} = \{a + nb | n \in \mathbb{Z}\}$, for $b > 0$.
We will now define a topology on $\mathbb{Z}$. We let sets O be open, if they are either empty, or if for each $a \in O$, there exists a $b > 0$ such that $N_{a,b} \subseteq O$.
We now verify that arbitrary unions of open sets are open. $\bigcup_{i \in I} O_i$ is either empty (in which case it is open), or for each $a \in \bigcup_{i \in I} O_i$, we pick an i such that $a \in O_i$, and consider a $b > 0$ for which $N_{a,b} \subseteq O_i$: by choosing this $b > 0$, we have $N_{a,b} \subseteq \bigcup_{i \in I} O_i$.
Next, we verify that intersections of open sets are open. Unless $O_1 \cap O_2$ is empty (in which case it is open), for each $a \in O_1 \cap O_2$, we obtain $b_1 > 0$ and $b_2 > 0$ so that $N_{a,b_1} \subseteq O_1$ and $N_{a,b_2} \subseteq O_2$. We my conclude by showing that $N_{a,b_1 b_2} \subseteq O_1 \cap O_2$.
Indeed, to see that wlog $N_{a,b_1 b_2} \subseteq O_1$, we note that $N_{a,b_1 b_2} \subseteq N_{a,b_1}$: terms of form $a + nb_1 b_2$ may be written in form $a + mb_1$ with $m = nb_2$.

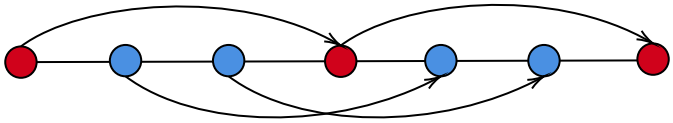
We make two remarks about this topology:

- Open sets are infinite, unless they're empty:
  indeed, for some $a \in O$, there exists a $b > 0$ such that $N_{a,b} \subseteq O$, and $N_{a,b}$ is infinite.
- The sets $N_{a,b}$ for $b > 0$ are closed:
  indeed, its complement $\mathbb{Z} \backslash N_{a,b}$ may be expressed as a union of open sets $\bigcup_{i=1}^{b-1} N_{a+i,b}$.
  To see that the sets $N_{a,b}$ are open, note that for each of their elements $a + nb$, we have $N_{a+nb,b} \subseteq N_{a,b}$.
  To see that an element $m \in \mathbb{Z} \backslash N_{a,b}$ is in $\bigcup_{i=1}^{b-1} N_{a+i,b}$, let i be the remainder of $m - a$ by b.

The link to primes is now made, as follows.
The fact that number m has prime divisor p now rephrases as $m \in N_{0,p}$. Seeing as all integers but 1 and -1 have prime divisors, we have $\mathbb{Z} \backslash \{-1, 1\} = \bigcup_{p \in P} N_{0,p}$, where P denotes the set of primes.
Now, if P were finite, then $\mathbb{Z} \backslash \{-1, 1\}$ would be closed, as a finite union of finite sets.
Then its complement $\{-1, 1\}$ would have to be open, yet this is impossible, as it's neither empty, nor infinite, a necessary condition for non-empty sets to be finite, in our topology.
Thus the set of primes P must be infinite. □

In our formalisation, we first define the two-way infinite arithmetic progressions N.
We then define our topology as an `instance`, by providing the property for a set to be open in field `is_open`, showing that the universal set is open in field `is_open_univ`, showing that intersection of open sets are open in `is_open_inter` and showing that arbitrary unions of open sets are open, in `is_open_sUnion`.

There are multiple steps of the proof we may prove individually, and we start with `N_as_a_comlpement`, where we show that $\mathbb{Z}\backslash N_{a,b} = \bigcup_{i=1}^{b-1} N_{a+i,b}$. We will rapidly illustrate its idea:



This illustrates an example where $b = 3$. The dots represent a segment the integers, where red dots belong to $N_{a.b}$, so that blue dots correspond to the complement. the complement can be represented as a union of progressions. To determine which progression an integer m of the complement is in, we consider remainder of $m - a$ by b, which corresponds to its rank between the red dots, counted from the left.

The formalisation went well, and we collect some of its specificities.
First we make use of remainders in the Euclidean division for *integers*. They are non-negative:

Remaniders are non-negative
`int.mod_nonneg`

If the dividend is non-zero, the remainder in non-negative.
In code: `∀ (a : ℤ) b : ℤ, b ≠ 0 → 0 ≤ a % b`

Another remarkable fact is the use of the tactic `linear_combination`, which spares us quite a tedious computattion in the claim we name `target`.

We use this lemma in `N_closed`, where we show that the sets $N_{a,b}$ are also closed.
We shall record two topological facts from the formal proof:

Topological facts
`is_open_compl_iff`
`is_open_bUnion`

The first states that a set is closed iff its complement is open.
The second states that union over a family f of open sets is open.
In code, respectively:
`is_open s^c ↔ is_closed s`
`(∀ (i : β), i ∈ s → is_open (f i)) → is_open (⋃ (i : β) (H : i ∈ s), f i)`

Next, we show another component of the proof in `two_units_not_open`: the fact that $\{-1, 1\}$ isn't open, or equivalently, that $\mathbb{Z}\backslash\{-1, 1\}$ is closed. Here, the formal proof differs a little from our informal one. We avoid mentioning infinity, showing instead that, assuming for contraction that $\{-1, 1\}$ is closed, if there were a $b > 0$ such that $N_{a,b} \subseteq \{-1, 1\}$, then $1 + b > 1 > -1$ would have to be in $\{-1, 1\}$.

We now show that $\mathbb{Z}\backslash\{-1,1\} = \bigcup_{p\in P} N_{0,p}$ in `univ_sdiff_units_as_prime_union`.
To get prime divisors of integers, we now use the following, which makes use of the absolute value of an integer:

Prime divisors of integers
`int.exists_prime_and_dvd`

If an integers absulte value isn't 1, it has a prime divisor.
In code : `∀ {n : ℤ}, n.nat_abs ≠ 1 → (∃ (p : ℤ), prime p ∧ p ∣ n)`

An important aspect is that this absolute value has natural numbers as their output type. Transitioning between integers and the absolute value is possible with lemmata such as `int.nat_abs_eq_iff : a.nat_abs = n ↔ a = ↑n ∨ a = -↑n`, or `int.prime_iff_nat_abs_prime : prime k ↔ nat.prime k.nat_abs`.

We may now show the infinitude of primes in `fifth_proof`.
The last topological fact we need for our proof is:

Union of closed sets
`is_closed_bUnion`

A finite union of closed sets in closed.
In code:
`s.finite → (∀ (i : β), i ∈ s → is_closed (f i)) → is_closed (⋃ (i : β) (H : i ∈ s), f i)`

This allows us to conclude quite effortlessly.

2 Chapter 11: Lines in the plane and decompositions of graphs

We have formalised the theorems labeled 1, 2 and 3, from the correspponding chapter in "Proofs from THE BOOK". Theorem 1 is formalised in file `FormalBook_Ch11_LinesInThePlane_SylvesterGallai`, and theorems 2 and 3 are contained in file `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry`. Theorem 2 makes use of theorem 1, so that we impor the first file in the latter.

2.1 The Sylvester-Gallai theorem

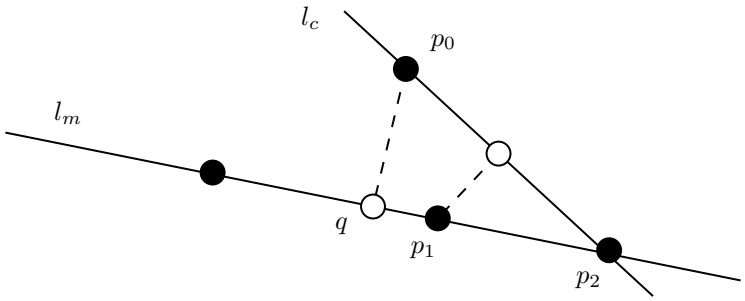
Refer to file `FormalBook_Ch11_LinesInThePlane_SylvesterGallai`.

We want to formalise the following theorem:

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 `line_nonempty`, complete, in `line_complete`, and convex, in `line_convex`. These conditions are necessary for us to define the orthogonal projection of a point on such a line, using `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 also 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. Note the nomenclature *"smul"* for scalar multiplication, denoted by the dot symbol.

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. Since this principle is the main focus of the next section, we will discuss it only semi-formally here. 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.

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 speak of a `finset.univ` for it.

Next, we move to our lemma `point_line_finset_nonempty`. Here, we show that if the points of P aren't 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 key existence result:

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 $\sqcap$ 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 based on the axiom of choice:

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

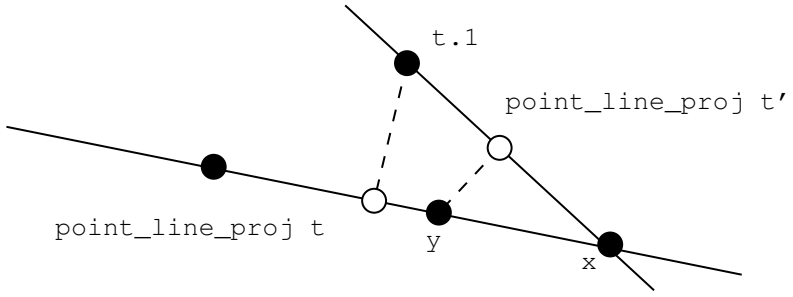
If we know that something of type T exists, which satisfies predicate p, then we may refer to it with `classical.some`, and use `classical.some_spec` to certify that it satisfies the predicate p.
In code, respectively:
`∏ {α : Sort u} {p : α → Prop}, (∃ (x : α), p x) → α`
`∀ {α : Type} {p : α → Prop} (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 `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

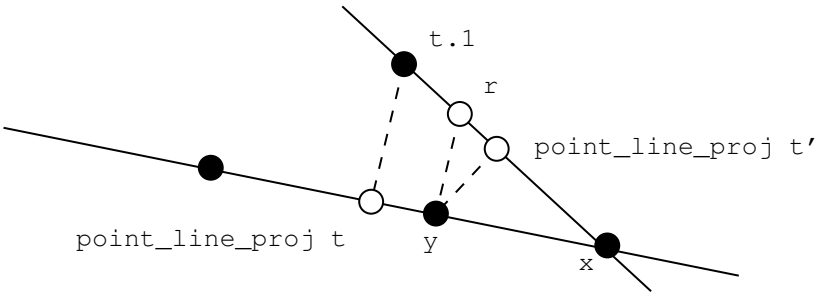
However, we have found a way to handle this constraint, in our lemma `SG_Pythagoras_workaround`, which uses:

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 charcterisation of minimisers, and the polarisation identity, 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 charcterisation of minimisers provides $\langle y-s, x-s\rangle \leqslant 0$ and $\langle y-s, y-x\rangle \leqslant 0$, 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.

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'`.
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 isn't the case for 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.
This concludes on of the lengthier formal proofs of this thesis.

2.2 Incidence geometry and graph decompositions

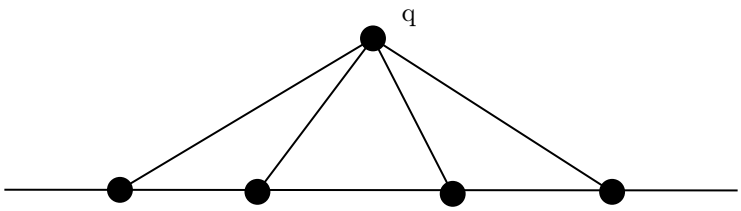
Refer to the file names `FormalBook_Ch11_LinesInThePlane_IncidenceGeometry`.

First, we want to formalise the following:

The Erdös-de Bruijn theorem

For a set P of points, we consider the set L of line passing through at least 2 points of P.
If not all point of P are on the same line, then $|P| \leqslant |L|$.

Proof: The proof is by induction on the size of P.
For less then 3 points, the points are aligned, so these cases do not apply, and for 3 unaligned points, we obtain one line per pair of points, so that the theorems statement reduces to $3 \leqslant 3$.
For the induction step, we start by using the Sylvester-Gallai theorem to obtain points p and q of P, which span a line that contains no other of the $n+1$ points of P. We consider the points of $P\backslash q$. If they are not aligned, the induction hypothesis provides us at least n lines. None of these lines may be the one spanned by p and q, as this would lead to that line containing at least 3 points of P, contradicting the property bestowed to it by the Sylvester-Gallai theorem. Hence, in this case, he have at least $n+1$ lines, and the step is proven.
If, on the other hand, the points of $P\backslash q$ are aligned, we have the following situation:



Note that q may not be on the line the points of $P\backslash q$ are aligned on, for otherwise, all points are of P are aligned. As the figure displays, in the "pencil" configuration, we get n lines passing through q, and the additional line the points of $P\backslash q$ are aligned on, so that the step follows in this case too. □

We have formalised this in the form of `theorem_2` in our file. First, we define the set L as the image of pairs of distinct points of P under the map that associates to such pairs of points the line that they span:

```
def line_set (P : finset E) :=
  finset.image
    (λ p : P × P, line (p.1 : E) (p.2 : E))
    (finset.filter (λ p : P × P, p.1 ≠ p.2) finset.univ)
```

We now move to the proof of `theorem_2`. Lean offers a particular induction principle for finite sets, that eases arguments in which a property is extended from a strict subset, such as from $P\backslash q$ to P in our informal proof.

Strong induction on finsets
`finset.strong_induction_on`

If a property of a finite set can be deduced from that of a proper subset, then it is true for any finite set (of elements of the given type).
In code: ∀(s : finset α), (∀ (u : finset α), (∀ (t : finset α), t ⊂ u → p t) → p u) → p s

In the notation of our formal proof, S represents P, u represents q and T represents $P\backslash q$. We then perform the case disjunction of whether the points of T are aligned. In the case that they are we `specialize` the induction hypothesis to T, and use an auxiliary lemma `list_set_lemma` to shorten the main proof.

The lemma `list_set_lemma` states that in our context, the size of the line set of $P\backslash q$ is strictly less then that of the line set of P. We show this with `finset.card_lt_card`, which reduses the problem to showing that the line sets are proper subsets, which using `finset.ssubset_iff`, reduces further to finding a line in the line set of P, which isn't in $P\backslash q$ and showing the the lines of $P\backslash q$ are among those of P. The line we will use is the one spanned by p and q.

Returning to the proof of `theorem_2`, we now discuss the case where all points of $P\backslash q$ are aligned, say on a line spanned by `a` and `b`. We define the n lines passing through q in the form of `L`. Over the course of 3 claims internal to the proof, we show that line spanned by `a` and `b` is not contained in `L`, that the "pencil", made of the line spanned by `a` and `b` and those of `L` are a subset of the line set of P, and that there are as many lines in `L` as there are in `T`. We then combine bounds and equalities to conclude with the inductive step.

Next, we formalise the following theorem, which generalises the previous to an abstract context:

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, far 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 cionsider one more lement 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 asumption, 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 $\dfrac{1}{m(|X| - |A_i|)} < \dfrac{1}{|X|(m - r_x)}$, 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:

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

□

We have formalised it as `theorem_3`. A remark relevant in the optic of publication on mathlib is that the assumption $0 < m$ is superfluous. Indeed, X has (at least) 3 elements, hence a pair, hence some A_i containing the pair, so that assumption `(hm : 0<m)` is actually implied by assumption `h` and `hX`.

We shall not discuss the formalisation in detail, so as to gain space to meet the upper-bound on the size of a thesis imposed by study-regulations. The formalisation uses no new area of mathlib, or technique, that may have been of particular interest at this stage of the thesis. We will merely comment on the fact that the proof is described as "almost one-line" in "Proofs from THE BOOK", whereas our formalisation required around 470 lines.
This serves as an example of how short straightforward arguments can require a much longer formalisation.

We merely record the structure of our formalisation. In `theorem_3_ub`, we show that assuming that $r_x = m$ is impossible, and in `theorem_3_lb`, we show that when $x \notin A_i$, then $r_x \geqslant |A_i|$. The lemma `splitter` proves $1 = \sum_{x \in X} \frac{1}{|X|}$ and the lemma `sum_rel` handles the double-counting: both contain important mathlib lemmata linking sums, index sets, and the sizes of the latter.

We apply the previous theorem to show the corollary in graph theory that follows.

Decomposition into cliques

If we decompose a complete graph K_n, for $n \geqslant 3$, into m cliques different from K_n, such that every edge is in a unique clique, then we have $m \geq n$.

Proof: We apply the Motzkin-Conway theorem as follows: we let X be the vertices of K_n and A_i be the vertices of clique i. We must verify that to each pair of vertices, so to each pair of elements of X, we may associate a unique A_i. Seeing as in K_n each pair of vertices is associated to an edge, and by the fact that the A_i from a decomposition of K_n, each edge is in a unique clique, the desired condition is satisfied. □

We have given two versions of this theorem and proof, in the form of `clique_decomposition` and `clique_decomposition'`, which differ in the way the notion of a clique as a subgraph is interpreted.

In both versions, the notion of decomposition is described as follows. We consider the indexed family of subgraphs `(D : fin m → subgraph (complete_graph V))`, and add hypothesis `h`, which states that for each edge of K_n, there is an index i such that the subgraph D_i contains this edge, and such that any D_j containing it, we must in fact have $j = i$, so as to have uniqueness. Note that we coerce subgraphs using `subgraph.spanning_coe`, so as to get an edge set of type V, compatible with edges of K_n, whereas using `subgraph.coe` would have required us coerce the edge set, a task for which there does not seem to exist support, to our knowledge.

The structure of the proof of both versions is the same. As in the informal proof, we let X be the universal set of the type V of vertices, and define `set A := (λ i : fin m, (D i).verts.to_finset)`. The proof consists mostly of definitional rewrites, using for example `mem_edge_set`. Only the last code block, which corresponds to uniqueness of membership in the decomposition, differs in both proofs.

In the first version `clique_decomposition`, we initially tried to express that the D_i are cliques through `(subgraph.coe (D i)) = complete_graph (D i).verts`. This definition is valid, but we could not complete the proof with it. Instead, we developed a similar assumption, in the form of `hD_2`. There, we consider D_i as a subgraph over the whole type of vertices V, and set it equal to the graph that consists of the clique on the vertices of D_i, and isolated vertices for the rest of the vertices of K_n. We build the latter graph from its adjacency relation, as is the default way to construct graphs in mathlib.

In the second version, `clique_decomposition'` we express that the D_i are cliques with the mathlib definiton:

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

This is a property of a set of vertices of the graph. We use it in assumption `hD_2`. There we tried to use `complete_graph V` instead of `subgraph.spanning_coe (D i)`, so as to highlight that D_i is a clique of K_n, but we could only complete the proof with the latter version.
In the proof we use the supporting lemma `is_clique_iff` of the library.

As it may have transpired from our exposition, we are not satisfied with our formalisation for this corollary on graph decompositions. 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` we mentioned above. We do not know if the transitions between all these concepts is well supported by mathlib, and which is the most adequate to use. In our defense, there is as of now no practical documentation available. If the content of this thesis is ported to Lean 4, and we attempt to publish it on mathlib, a discussion with the mathlib maintainers will be insightful on this matter.

3 Chapter 27: Pigeon-hole and double counting

The corresponding chapter of "Proofs from THE BOOK" contains a collection of beautiful uses of two fundamental tools from combinatorics: the pigeon-hole principle, and double counting.

A general version of the pigeonhole principle is present in mathlib in the for of:

General pigeonhole principle
`finset.exists_lt_card_fiber_of_mul_lt_card_of_maps_to`

Given a map f mapping pigeons from a finite set s to a finite set t of pigeonholes, so that $|t|n < |s|$ for some natural number n, we may conclude with the existence of a pigeonhole $y \in t$ that contains at least n pigeons.
In code: `(∀ (a : α), a ∈ s → f a ∈ t) → t.card * n < s.card → (∃ (y : β) (H : y ∈ t), n < (filter (λ (x : α), f x = y) s).card)`

A special case is mathlib's:

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 pigeonhole principle will be put to use in sections *"Numbers"*, *"Sequences"* and *"Sums"*.

The double counting technique is also contained in mathlib. 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's 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`

Double counting will be put to use in sections *"Numbers again"* and *"Graphs"*.

3.1 Numbers

Refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_numbers`.

Our first application of the pigeonhole principle will be:

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's 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. □

The proof is formalised in our files `claim_1`.
We first show that a number and its successor are coprime, in `succ_coprime` using:

Coprimality facts
`nat.coprime_self_add_left`
`nat.coprime_one_left`

First, n and m are coprime iff $m+n$ and m are coprime. Second, 1 is coprime with any number.
In code, respectively:
`∀ {m n : ℕ}, (m + n).coprime m ↔ n.coprime m`
`∀ (n : ℕ), 1.coprime n`

Now we turn to the claim.
The map we use to map pigeons to pigeonholes is $n \mapsto \lfloor \frac{n+1}{2} \rfloor$. The reader may convince themselves that 1 and 2 have image 1, and $2n-1$ and $2n$ have image n. We apply the pigeonhole principle to this map in our claim `Lem1`, where the map is called `group_fn`. Recall that for natural numbers, In Lean x/y denotes the quotient of x by y, which corresponds to $n \mapsto \lfloor \frac{x}{y} \rfloor$.
We apply the principle by showing the mapping condition for the principle in claim `map_condition`, which requiries some . After applying the principle in form `exists_ne_map_eq_of_card_lt_of_maps_to`, we're left to show the inequality $|[n]| < A.card$, where A is our set of pigeons.

Next we use the two numbers obtained from the principle as candidates for our coprime numbers.
We now have to prove that these number are successive ones, and settle which is the successor among the two.
In claim `Lem2` we show that the numbers must have different remainders in the division by 2, using mathlib's version of Euclidean division, linking quotient x/y and remainder $x\%y$:

Euclidean division
`nat.div_add_mod`

If q is the quotient in the division of a by b, and r the remainder, then $a = bq + r$.
In code: `∀ (m k : ℕ), k * (m / k) + m % k = m`

We use `Lem2` in or application of the `wlog` tactic, which allows us to settle, *w*ithout *l*oss *o*f *g*enerality, which of the numbers is the successor: we set it to b. We take note of `nat.coprime_comm`, which states that a and b are coprime iff b and a are (symmetry), in the `wlog` code block.

To obtain the fact that the remainder of $b+1$ by 2 is 1, we make use of the `interval_cases` tactic to disjoin among the possible values of $(b+1)\%2$, which we upper-bound with `nat.mod_lt : 0 < y → x % y < y`.
The first case is shown impossible with `nat.not_lt_zero : ∀ (a : ℕ), ¬a < 0`.
In the second case, we put our plan to action, using our `succ_coprime`, and conclude the proof.

Next, we give a little more complicated application of the pigeonhole principle:

Appearance of divisors

No matter how we pick $n+1$ numbers among the numbers $1, 2, ..., 2n$,
there will always be two picked number for which one divides the other.

Proof: Each number from $1, 2, ..., 2n$ can be factored to have form $2^k m$, where k and m are naturals and m is odd: indeed, we let k be the valuation of 2 in that number's prime decomposition.
Now, m can't exceed the number it is a factor of, so it cannot exceed $2n$ in particular.
Thus, m is constrained to be an number odd number ranging in $1, 3, ..., 2n-1$, of which there are n.
Therefore, if we map the $n+1$ pigeon-numbers of $1, 2, ..., 2n$ to the odd part of their prime decomposition, which represent the holes, we obtain two numbers with the same odd part of their prime decomposition.
Concretely, we get a and b such that $a = 2^{k_a} m$ and $b = 2^{k_b} m$ for odd part m.
Then, if wlog $k_a \leqslant k_b$, we have $a|b$ as $b = 2^{k_b - k_a} a$, and we found our desired pair. □

We shall start our formalisation by showing the decomposition, in `decompo_lemma`. At the time, we were not aware of mathlib's prime factorisation support, to be found in folder `data.nat.factorisation`. We are also unsure whether the use of prime factorisation would have made the proof shorter then by defining the k in decomposition $n = 2^h m$ to be maximum for this property, so as to argue that the corresponding m is odd, which is the proof strategy we embarked on.
Thus, in `decompo_lemma`, for a given $a \in [2n]$ we define `facSet`, the set of q such that $2^q | a$, and define k to be its `finset.max'`, and m to be the rest of the factorisation, so that $a = 2^q m$. After proving that $m \in [2n]$, we prove that it is odd. We do so by contradiction, as assuming that $m = 2p$ implies that $a = 2^{k+1} p$, which contradicts the maximality of k. Key mathlib lemmata used here are `even_iff_two_dvd : even a ↔ 2 ∣ a`, `finset.le_max'`, and the handy `nat.not_succ_le_self : ¬n.succ ≤ n`.

After a quick size computation in `size_lemma`, which proves that the size of odd numbers in $[2n]$ is n, by using `card_eq_of_bijective` and displaying the bijective map $x \mapsto 2x+1$ which maps the numbers of $0, 1, ..., n-1$ to the odd numbers of $[2n]$, we move to the proof of the result we set out to formalise, in `claim_2`.

To define the map required by the pigeonhole principle, which to a given a associates the m from the decomposition of `decompo_lemma`, we have to eliminate the existential quantifier from this statement. This is done with `nat.find`, a special case of `classical.some` for natural numbers.

In addition, we are faced with the following problem: the pigeonhole principle, in the form of `exists_ne_map_eq_of_card_lt_of_maps_to`, does not use a pi-typed function, so that the function must be defined on the whole type. In our case, the type is that of the natural number. We must however make use of the domain $[2n]$, so as to be able to justify that the output is an odd number in $[2n]$, using `nat.find_spec`, a special case of `classical.some_spec`. Therefore, when defining the pigeonhole-map `f`, we extend it to $\mathbb{N}$ by setting its output to 0 for inputs outside of $[2n]$. The rest of the formal proof consists mostly of algebraic manipulations. We performed a case disjunction on whether $k_a \leqslant k_b$: using the `wlog` tactic to assume this particular case, may be an alternative.

Seeing as we made use of conditioning, we will state how it is handled in practice, to conclude our discussion.

Rewriting conditioned functions
`if_pos`
`dif_pos`

If the condition is satsified, we may replace the conditioned function with its output for the positive case. The versions differ by the case of whether the outputs are function *d*epending on the condition.
In code, repectively:
`if_pos : ∀ {c : Prop} [h : decidable c], c → ∀ {α : Sort u} {t e : α}, ite c t e = t`
`dif_pos : (hc : c) {t : c → α} {e : ¬c → α}, dite c t e = t hc`

3.2 Sequences

Refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_sequences`.

We shall now see an application of the generalised pigeonhole principle, in the following.

Erdös-Szekeres theorem

Among any $a_1, a_2, ..., a_{mn+1}$ distinct real numbers, we may find either a subset of indices $i_1 < ... < i_{m+1}$ such that $a_{i_1} < ... < a_{i_{m+1}}$, or a subset of indices $i_1 < ... < i_{n+1}$ such that $a_{i_1} > ... > a_{i_{n+1}}$, or both.

Proof: To each index i, we may associate the length t_i of the longest increasing sub-sequence starting at i, in the sense that t_i is maximum for the property of finding $i_1 < ... < i_{t_i - 1}$ such that $a_i < a_{i_1} < ... < a_{i_{t_i - 1}}$. If we find an i such that $t_i \geqslant m + 1$, then we have satisfied the first case of the statement. Otherwise, we gain the assumption that for all i, we have $t_i \leqslant m$.
In such a case, we consider the $mn + 1$ numbers t_i as pigeons, which fit into m holes. By the generalised pigeonhole principle, there are at least $\frac{mn}{m} + 1 = n + 1$ pigeons in one of the holes. Ordering the indices corresponding to the same hole to be $j_1 < ... < j_{n+1}$, the latter translates to the fact that for each of these j_i, the longest increasing subsequence starting from them all have the same length, denoted τ.
We will now show that we must have $a_{j_i} > a_{j_{i+1}}$ for these indices, so that they constitute the second case of the claim. Indeed since the number are distinct, assuming otherwise would be assuming $a_{j_i} < a_{j_{i+1}}$. This however implies that we may extend the longest increasing subsequence starting at j_i. Indeed, if we consider the sequence of length τ starting at j_{i+1}, then pre-pending j_i to the sequence produces a increasing sequence, of length $\tau + 1$. Yet, since the longest increasing sequence at j_i is τ, this contradicts maximality.
Thus, we must be in the second case claimed by the theorem. □

We start our formalisation by defining the notions of increasing and decreasing subsequences.
We let these notions be properties of subsets of indices I and we include the starting index s in this property:

```
def is_increasing_subseq (I : finset ℕ) (a : ℕ → ℝ) (s : ℕ) : Prop :=
  (∀i∈I,∀j∈I, (i<j) → a i < a j) ∧ (∀j∈I, s≤j) ∧ (s∈I)

def is_decreasing_subseq (I : finset ℕ) (a : ℕ → ℝ) (s : ℕ) : Prop :=
  (∀i∈I,∀j∈I, (i<j) → a i > a j) ∧ (∀j∈I, s≤j) ∧ (s∈I)
```

We shall define the t_i from the infromal proof as follows:

```
def longest_incr_subseq_len
  (I : finset ℕ) (a : ℕ → ℝ) (i : ℕ) (i_I : i∈I) : ℕ :=
  finset.max' ((range (I.card +1)).filter (λ t, (∃J⊆I, (J.card = t) ∧
  (is_increasing_subseq J a i)))) (lengths_nonempty I a i i_I)
```

Here, I will be the index set of the sequence a. In the formalisation of the theorem, we will have $I = [mn + 1]$. Mathlib publication guidelines would however recommend us letting I be any finset of size $mn + 1$ over any type with an total order $<$. Next, i will be the start of the subsequence. The possible lengths for subsequences range in $0, ..., |I|$, and we filter this range depending on whether a subset of indices corresponding to an increasing subsequence satring at i has this length. The maximum of this set of lengths will then be t_i. Since we are using `finset.max'` to define the maximum, we need to prove non-emptyness, in `lengths_nonempty`. The latter follows from the fact that i, taken on its own ,constitutes a increasing subsequence.

With respect to our file, we skip ahead to the statement of the Erdös-Szekeres theorem, named `claim`. We show a mildly stronger result then the actual statement we gave: we show that the increasing sequence has length *at least* $m + 1$ and that the decreasing sequence has length *at least* $n + 1$, both of which imply the existence of monotone subsequence of the exact lengths.
We start the proof with the following logic lemma, which captures the case disjunction more efficiently:

A logic lemma
`or_iff_not_imp_left`

Showing "a or b" is equivalent to showing that "if not a, then b".
In code: `∀ {a b : Prop}, a ∨ b ↔ ¬a → b`

We let `f` be the t_i, as a function, from our informal proof, which we will use with the generalised pigeonhole principle. Just as in the previous subsection, the principle does not make use of pi-type, so we extend `f` to all of the naturals via conditioning.

We apply the generalised principle `finset.exists_lt_card_fiber_of_mul_lt_card_of_maps_to` after having shown its necessary conditions that we called `map_cond` and `size_cond`, to get the "hole" j, corresponding to the τ from our informal proof, that contains at least $n + 1$ pigeons. We denote the set of the latter by I, and let i be its minimum. We then `use` i as the start of the decreasing sequence, and I as its index set.
The fact that I had size at least $n + 1$, which is the required goal of the last code block, is given directly from the generalised principle. The proof that the sequence indexed by I is decreasing revolves around the auxiliary `lisl_non_extendable`, that we shown earlier in the file and that we now discuss.

In this lemma, we prove that for indices i and j, where $j < i$, such that the lengths of the longest increasing subsequence starting at each of these indices are equal, we have $a_j > a_i$. In the notation of this informal proof, we show the step corresponding to $a_{j_i} > a_{j_{i+1}}$. As in the informal proof, this is shown by contradiction, where we may `replace` the contradictory hypothesis with $a_j < a_i$, as the numbers are assumed to be distinct. We then consider the indices achieving the longest increasing subsequence, starting from i, which we name `index_seq`.
In our claim named `extend`, we show that `insert j index_seq`, which should be read as a more Lean-idiomatic version of $\{j\} \cup$ `index_seq`, is a increasing subsequence starting at j. To see this, we must take arbitrary members of this set, disjoint cases on whether they are j or not, and use the appropriate assumptions to conclude in each case. Throughout the proof, we make use of support for `finset.insert`:

Support for `finset.insert`
`finset.mem_insert`
`finset.mem_insert_self`
`finset.insert_subset`
`finset.card_insert_of_not_mem`

In code, respectively:
```
∀{s : finset α} {a b : α}, a ∈ insert b s ↔ a = b ∨ a ∈ s
∀(a : α) (s : finset α), a ∈ insert a s
insert a s ⊆ t ↔ a ∈ t ∧ s ⊆ t
a ∉ s → (insert a s).card = s.card + 1
```

We have and will not discuss this type of support for other parts of the thesis. Note, however, that most of the proofs we build are made up in great parts from this type of support.

Moving further in the proof of `lisl_non_extendable`, we show in the claim named `contra_pre` that the size of `insert j index_seq` is among the lengths of the increasing subsequences starting at j, and we use it in claim `contra_pre_2` to show that this size must be less then the maximum length of such subsequences. Then, using `finset.card_insert_of_not_mem` and the fact that the length of `index_seq` is equal to the latter maximum, we derive a contradiction from the inequalities on sizes/lengths we set up so far.

This concludes our discussion of the proof of `lisl_non_extendable`, and therefore also that of theorem `claim`.
In "Proofs from THE BOOK", the section contains a beautiful application of the Erdös-Szekeres theorem to a problem on permutations. We did not fromalise it due to lack of time. Mathlib offers support for permutations in the form of the folder `group_theory.perm`. However, the formalisation of arguments such as "just delete $n+1$ in a representation of K_{n+1}" has, as of now, no support in mathlib, and formalising this support may require as much time and effort as formalsining the result on graph dimensions that follows the exerpt we gave from "Proofs from THE BOOK".

3.3 Sums

Refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_sums`.

We now show one last application of the pigeonhole principle.

Vázsonyi-Sved theorem

Consider n integers in a sequence $a_1, ..., a_n$, that are not necessarily distinct. Then, we may find a set of consecutive numbers who's sum is divisible by n. That is, we may find indices $k < l$ such that $n | \sum_{i=k+1}^{l} a_i$.

Proof: We consider the pigeon-map $f(m) = \left(\sum_{i=1}^{m} a_i\right) \% n$, associating the number of terms m in the partial sum of the number a_i to its remainder by n. We define it on $0, 1, ..., n$, with convention $\sum_{i=1}^{0} a_i = 0$.
Since its images range in $0, 1, ..., n-1$, we may use the pigeonhole principle to deduce that for some indices $k < l$, we have $\left(\sum_{i=1}^{l} a_i\right) \% n = \left(\sum_{i=1}^{k} a_i\right) \% n$. This rewrites to $\left(\sum_{i=k+1}^{l} a_i\right) \% n = 0$, and hence $n | \left(\sum_{i=k+1}^{l} a_i\right)$. □

We have formalised this result under the name `claim` in our file. It is preceded by two technical lemmata we named `tec_mod` and `tec_sum`. In our formalisation, we chose relatively arbitrarily to use `int.nat_mod`, a remainder who's input types are integers, and who's output type is a natural number. The lemma `tec_mod` describes the rewrite step for equalities of remainders from our informal proof. It is based on:

Integer remainders and subtraction
`int.sub_mod`

In code: `∀ (a b n : ℤ), (a - b) % n = (a % n - b % n) % n`

where `%` denotes `int.mod`, an integer valued remainder for integer inputs.

Similarly, `tec_sum` describes the rewrites for sums. An interesting feature of its proof is the sue of `disj_union` to describe the disjoint union of sets, and the following lemma, which allows to split sums on disjoint unions of index sets.

Splitting sums
`finset.sum_disj_union`

For disjoint sets A and B, we have $\sum_{x\in A\cup B} f(x) = \sum_{x\in A} f(x) + \sum_{x\in B} f(x)$.
In code: `(h : disjoint s1 s2), ∑ (x : α) in s1.disj_union s2 h, f x = ∑ (x : α) in s1, f x + ∑ (x : α) in s2, f x`

The formalisation of the theorem in `claim` is relatively straightforward.
We define `f` as in the informal proof, where we do not need to artificially extend the function, as its output does not depend on the domain $0, 1, ..., n$ that we will use for the application of the pigeonhole principle.
To settle which of the indices is greater, the assumption $k < l$ in the informal proof, we make use of `lt_by_cases : ∀ (x y : α) P : Prop, (x < y → P) → (x = y → P) → (y < x → P) → P` instead of the `wlog` tactic, as the latter fails, for reasons we could not make out, as can be seen by un-commenting the line above the use of `lt_by_cases`.

We conclude this section with the following mathlib take-away, used in our proof :

Remainders and divisibility
`int.dvd_of_mod_eq_zero`

If the remanider of b by a is zero, then a divides b.
In code: `∀ {a b : ℤ}, b % a = 0 → a ∣ b`

3.4 Numbers again

Refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_numbers_again`.

We shall now see a first use of double counting:

Average number of divisors

Denote by $d(i)$ the number of divisors of natural number i. Then we may bound their average with:

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

Proof: Since $d(i) = |\{j \leqslant n : j|i\}|$, we may use double counting to write $\sum_{i=1}^{n} d(i) = \sum_{j=1}^{n} |\{i \leqslant n : j|i\}|$, by considering divisibility as a relation between sets $[n]$ and $[n]$.
Now, $\{i \leqslant : j|i\}$ is the set of multiples of j in $[n]$. They are easier to count: $|\{i \leqslant n : j|i\}| = \left\lfloor \frac{n}{j} \right\rfloor$.
Hence, we may write $\sum_{i=1}^{n} d(i) = \sum_{j=1}^{n} \left\lfloor \frac{n}{j} \right\rfloor$. The bounds from the statement follow from the inequality $x - 1 \leqslant \lfloor x \rfloor \leqslant x$ and some algebraic manipulations. □

We start our formalisation with our `num_of_mult_le`, in which we elucidate the size of the number of multiples.

First we note that $\left\lfloor \frac{n}{j} \right\rfloor$ is the quotient of n by j, which we may simply write as `n/j` in Lean, including in our statement of `num_of_mult_le`. In that statement, we stated divisibility in terms of an equation directly (as $a|b \Leftrightarrow \exists c, b = ac$), so as to avoid rewrites. Our size computation will make use of:

Size computation lemma
`finset.card_image_of_injective`

The image under an injective map has the same size as the original set.
In code: `∀(s : finset α), function.injective f → (image f s).card = s.card`

We describe the set of mulitples of j in $[n]$ as the image of $\left[\left\lfloor \frac{n}{j} \right\rfloor\right]$ under map $x \mapsto jx$, so as to compute their size with this mathlib lemma. Indeed, if $k = jx \in [n]$, then equivalently, $x \in \left[\left\lfloor \frac{n}{j} \right\rfloor\right]$.

Next, in our lemmata `mult_rel` and `dvd_rel`, we express the set of multiples and divisors in terms of the support for double counting provided by mathlib, `bipartite_above` and `bipartite_below`. We then perform double counting in our lemma `main_result_pre`, using `sum_card_bipartite_above_eq_sum_card_bipartite_below`.

We rewrite the size of the multiples in the latter equality in our lemma `main_result_pre_rw`, and cast it to the rationals in `main_result_pre_cast`. Indeed, so far we have work with natural numbers only, but we will have to rationals in order to consider the average number divisors. The notation `((n/m : ℕ) : ℚ))` may be quite offsetting, however it is necessary, as `(n/m : ℚ)` would cause Lean to interpret this as a fraction. We want this to be the quotient, interpreted as a rational number, so that we use the previous notation.
We also note the support for showing quality of sums `finset.sum_congr : s1 = s2 → (∀ (x : α), x ∈ s2 → f x = g x) → s1.sum f = s2.sum g`.

We then show the upper-bound on the average number of divisors in our `upperbound`. The key ingredient is:

Inequality
`nat.cast_div_le`

We have $\left\lfloor \frac{a}{b} \right\rfloor \leqslant \frac{a}{b}$.
In code: `∀ {α : Type} [linear_ordered_semifield α] {m n : ℕ}, ↑(m / n) ≤ ↑m / ↑n`

Mathlib has support for the floor of a number, in the form of `int.floor`, and the inequality $\lfloor x \rfloor \leqslant x$ is present in form `int.floor_le`. Our choice to stick with coerced quotients was arbitrary. It simplified the step of computing the size of the set of multiples, but it requires us to show our own version of $x - 1 < \lfloor x \rfloor$, which is present in mathlib as `int.sub_one_lt_floor`.

We show our version of the latter inequality in our `lb_pre`. It consists of algebraic manipulations, that use `nat.mod_add_div` to introduce the remainder and `nat.mod_lt`, and facts about coercion of naturals.
We draw attention to `mul_div_cancel' : ∀ {G : Type} [comm_group_with_zero G] {b : G} (a : G), b ≠ 0 → b * (a / b) = a`. Indeed, we may define `(1 : ℚ)/(0 : ℚ)` in Lean. In fact, it is evaluable, and `#eval`uates to 0. In order to perform algebraic manipulation in the instances of multiplicative groups with 0, we have to justify that denominators are non-zero.

After a technical rewrite in `last_rw`, we prove the lower-bound from the main proof, in `lowerbound`. This is done mostly with casting lemmata and algebraic lemmata with sums. Finally, we merge all previous bounds and rewrites in `claim`, which states the formal version of the bounds on the average number of divisors.

We mention that we would be interesting to revisit this formalisation, relying on `int.floor` for a change.
To conclude this section, we take note of some take-aways reguarding manipulations with sums and coercion.

Manipulations with sums
`finset.sum_le_sum`
`finset.mul_sum`
`finset.sum_sub_distrib`

In code respectively:

- `(∀ (i : ι), i ∈ s → f i ≤ g i) → ∑ (i : ι) in s, f i ≤ ∑ (i : ι) in s, g i`
- `b * ∑ (x : α) in s, f x = ∑ (x : α) in s, b * f x`
- `∑ (x : α) in s, (f x - g x) = ∑ (x : α) in s, f x - ∑ (x : α) in s, g x`

Coercion of naturals
`nat.cast_inj`
`nat.cast_lt`

In code, respectively:

- `{m n : ℕ}, ↑m = ↑n ↔ m = n`
- `{m n : ℕ }, ↑m < ↑n ↔ m < n`

3.5 Graphs

Refer to the file named `FormalBook_Ch27_PigeonholeDoublecounting_graphs`.

The first application of double counting in graph theory given in the corresponding chapter of "Proofs from the BOOK" is already implemented in mathlib in the form of:

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 4-cycles, 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 neightbours 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. □

First we 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 proper 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`. 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 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, using 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 `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 to be proved have the same features, which we now comment on. Metaprogramming makes its first appearance 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. 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.

Next we define our double counting relation, stating that all (both) elements of the pair e are agjacent to u:

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

The next step in our proof 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 note 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'`. The first will compute the size through the use of finsets, whereas the second will compute it using the 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 on vertex of an edge of type `smy2`, 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 relevant lemmata:

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`

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`. 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 `sym2.card_subtype_not_diag : ∀ {α : Type}, fintype.card {a // ¬a.is_diag} = (fintype.card α).choose 2`.

We may then summarise the efforts mad 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 is put to use.
From here on, the proof is purely computational, 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.

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 who's 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 inquality 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 computational, so that we may state it without the actual numbers of interest for the theorem, in `max_edges_of_c4_free_Istvan_Reiman_pre`. 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're 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 is 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. For closing words, we choose to say that our calculations are not exemplary. We made use of algebraic tactics such as `cancel_denoms`, which cancels denominators in in-/equalities, yet this does not seem to shorten the calculation. In the `example` below the proof, we give an `example` of a use of the `ring` to derive the canonical form, though it would require us to to rewrite inequalities before and after this step.

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 two of them are 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 isn't 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:

- 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`.

Finally, 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. We will only collect some take-aways from the formal proof:

Support for spans
`submodule.mem_span_insert`
`submodule.span_induction`

We comment on the second one only: if a property p of a vector is true for 0 and is stable under addition and scalar multiplication, and is satisfied by any vector of a set S, then it is satisfied by any vector in $span(S)$. In code, respectively:

- `x ∈ submodule.span R (insert y s) ↔ ∃ (a : R) (z : M) (H : z ∈ submodule.span R s), x = a • y + z`
- `{p : M → Prop}, x ∈ submodule.span R s → (∀ (x : M), x ∈ s → p x) → p 0 → (∀ (x y : M), p x → p y → p (x + y)) → (∀ (a : R) (x : M), p x → p (a • x)) → p x`

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.