On Two Problems from Elementary Number Theory

Sanjay Adhith

November 2024

Contents

1 Abstract **4**

2 The Parity of Fibonacci numbers **5**

2.1 Formulating the theorem 5

2.2 Formal Statement of the theorem 5

2.3 Proof of forwards implication 6

2.4 First proof of the backwards implication 8

2.4.1 A slight refinement 10

2.4.2 A less slight refinement 11

2.4.3 Another refinement! 13

2.4.4 A proof by strong induction 15

2.4.5 Why this Proof was thrown out of The Book 15

2.5 Second proof of the backwards direction 16

2.6 What about the Odd Fibonacci numbers? 18

2.7 Lagniappe: An artful take on this problem 20

2.8 Vantage Loaf: Proof Sketch with Modular Arithmetic 23

3 Polygonal Numbers **24**

3.1 Identifying the theorem 24

3.2 Squares and Floors 25

3.3 On an exercise by Knuth 27

3.4 A Natural (Number) Deconstruction of Prof Danvy's Curiosa. . 30

3.5 Some geometric intuition 31

3.6 Lagniappe: Some additional series computed with floors 32

3.6.1 A conjecture on factorials 33

3.6.2 A conjecture on Möbius polynomials 33

4 Conclusion **35**

4.1 A Direct Communication 35

1 Abstract

There were anomalies everywhere, and the more I looked the more I saw!

— Stanford Pines
Gravity Falls, A Tale of Two Stans

In this project, two problems from elementary number theory are explored with the aid of the Coq Proof Assistant (tCPA). The first is concerned with the parity of Fibonacci numbers (is the fifth Fibonacci number odd or even?). The second is a discussion of a formula for computing polygonal numbers from Prof Danvy's *Summa Summarum*[1].

The prerequisites to follow the core body of this final project includes a basic familiarity with both the Coq Proof Assistant, the technique of mathematical induction, and high-school algebra.

To follow the digressions in the Lagniappes and Vantage Loaf, a basic knowledge of elementary number theory is required. However, these are in no way required to follow the main line of discussion and may be safely skipped over.

[1]Danvy, O. (2024). Summa Summarum: Moessner's Theorem without Dynamic Programming. Electronic Proceedings in Theoretical Computer Science, 413, 57–92. https://doi.org/10.4204/eptcs.413.5

2 The Parity of Fibonacci numbers

2.1 Formulating the theorem

The Fibonacci numbers begin with 0 and 1, and then each subsequent number is the sum of the previous two numbers. Let F_n denote the n-th Fibonacci number, where n is a natural number. The first problem is to try and identify when the terms of the Fibonacci numbers are even and when they are odd.

To help formulate a conjecture the Fibonacci numbers may be written down along with their parities. The bottom row in the below table considers the parities of the corresponding Fibonacci number.

F_n	1	1	2	3	5	8	13	21	34	55
n	1	2	3	4	5	6	7	8	9	10
p	o	o	e	o	o	e	o	o	e	o

This numerical evidence allows us to conjecture that F_n is even if and only if n is a multiple of three.

2.2 Formal Statement of the theorem

First, we define the Fibonacci numbers recursively as such:

```
Fixpoint fib (n : nat) : nat :=
  match n with
    0 =>
    0
  | S n' =>
    match n' with
      0 =>
      1
    | S n'' =>
      fib n' + fib n''
    end
```

```
end.
```

In this project, we are not concerned with matters of performance, so the naive implementation which is easy to reason about is chosen.

The corresponding fold-unfold lemmas are in the .v file and will be referred to in this report.[2]

To consider a more efficient implementation of the Fibonacci numbers (say, using two accumulators or with the closed form formula involving the golden ratio), it would suffice to prove that this more efficient implementation is equivalent to the implementation described in this report.

Now, the theorem is stated as such:

```
Theorem parity_of_fibonacci_numbers:
  forall n : nat,
    (exists k2 : nat, fib n = 2 * k2) <-> (exists k3 : nat, n = 3 * k3).
```

2.3 Proof of forwards implication

First notice that the pattern holds for every three numbers, so we use the `nat_ind3` induction principle to show that $P(n)$, $P(n+1)$ and $P(n+2)$ imply $P(n+3)$ and be done with it[3].

The base cases are trivial. So, we look at the induction step closely. Our hypotheses looks like this:

```
    n'' : nat
    IHn' :
    forall k2 : nat,
    fib n'' = 2 * k2 -> exists k3 : nat, n'' = 3 * k3
    k2 : nat
```

[2]Olivier, Danvy. (2022). *Fold–unfold lemmas for reasoning about recursive programs using the Coq proof assistant. Journal of Functional Programming, 32 Available from: 10.1017/S0956796822000107*

[3]The base cases would be $P(0)$, $P(1)$ and $P(2)$

```
H_k2 : fib (S (S (S n''))) = 2 * k2
============================
exists k3 : nat, S (S (S n'')) = 3 * k3
```

The job would be done if we could use the inductive hypothesis. But to get to the antecedent we need to do some work.

First, we begin by unfolding inside of `H_k2`, which (after some routine associative-commutative rewriting) gets us:

```
H_k2 : 2 * fib (S n'') + fib n'' = 2 * k2
```

Now, it should be obvious that `fib n''` must be even, but this is not a standard theorem that comes in tCPA, so we formulate it as such:

```
Lemma about_sum_of_even_numbers:
  forall a b c : nat,
    2*a + b = 2*c ->
      exists k : nat,
          b = 2 * k.
```

The proof is done by structural induction.

Jack: "Can't we just prove this by saying that there exists `c - a`"?

Joker: "Yes, this does work, you can indeed write something like

```
intros H_IHb'.
exists (c - a).
Search (_ * (_ - _)).
rewrite -> Nat.mul_sub_distr_l.
rewrite <- H_IHb'.
rewrite -> Nat.add_comm.
rewrite -> Nat.add_sub.
reflexivity.
```

. But it is nicer to do it structurally so that we do not have to resort to subtraction."

Joker (Confounded): "Pray tell."

Jack: "Let's just approach the problem using induction. Begin by inducting upon a and then leave c unqantified."

Joker: "Yes. Since you have the hypothesis `S (S (2 * a' + b)) = S (S (2 * c'))`, and then we have `a' + (a' + 0) + b = c' + (c' + 0)` in our hypotheses."

Joker: "Oh, so then, after some rewriting, we can use our induction hypothesis:

```
forall c : nat,
2 * a' + b = 2 * c -> exists k : nat, b = 2 * k
```

.

and conclude the proof!"

Using this lemma, using the `pose` tactic of tCPA, we then have:

```
H_n'' := about_sum_of_even_numbers (fib (S n'')) (fib n'') k2 H_k2 : exists k : nat, fib n'' = 2 * k
```

And then by we will be able to use the induction hypothesis to complete the proof.

2.4 First proof of the backwards implication

The backwards implication is now considered.

Again, we notice that the pattern only holds for every three numbers, so we can use `nat_ind3` to show that $P(n)$, $P(n+1)$ and $P(n+2)$ imply $P(n+3)$ and be done with it[4]

The base cases are trivial. So again, we look at the induction step closely. Our hypotheses

[4]Just as before, the base cases would be $P(0)$, $P(1)$ and $P(2)$

include $P(n)$, $P(n+1)$ and $P(n+2)$ is assumed, starting with what we want to prove we write down:

$$F_{n+3}$$

The natural thing to do next is to unfold the definition:

$$F_{n+3} = F_{n+2} + F_{n+1}$$

Since see no clear way to conclude that what we have is even right now, so let's unfold F_{n+2} again, since that's the only thing that we can unfold right now.

$$F_{n+3} = F_{n+1} + F_n + F_{n+1}$$

After rearranging some brackets, we have:

$$F_{n+3} = 2F_{n+1} + F_n$$

Now its obvious that F_{n+3} has to be an even number, because F_n is even by our induction hypothesis and $2F_{n+1}$ is also even since it is two times F_{n+1}.

```
rewrite -> fold_unfold_fib_SSS.
intros k3 H_up_three.
```

The goals are then transformed to become:

```
exists k2 : nat, 2 * fib (S n'') + fib n'' = 2 * k2
```

Now, we need to show that `fib n''` is twice of something, but our induction hypothesis only shows that `H_up_three : S (S (S n'')) = 3 * k3`.

We also have `IHn' : forall k3 : nat, n'' = 3 * k3 -> exists k2 : nat, fib n'' = 2 * k2`

It feels obvious to us that if `S (S (S n''))` is a multiple of three, then `n''` is a multiple of three.

```
Lemma shifting_by_threes:
  forall n k2: nat,
    S (S (S n)) = 3 * k2 ->
    (exists k3 : nat,
        n = 3 * k3).

Proof.
  intros n [ | k2'] H_k2.

  - rewrite -> Nat.mul_0_r in H_k2.
    discriminate H_k2.

  - rewrite <- thrice_S in H_k2.
    injection H_k2 as H_k2'.
    exists k2'.
    rewrite -> H_k2'.
    ring.
Qed.
```

Now the proof is done by combining the hypotheses as such.

```
    pose (shifting_by_threes n'' k3 H_up_three) as H_down_three.
    destruct H_down_three as [k3' H_down_three].
    pose (IHn' k3' H_down_three) as IHn'_down.
    destruct IHn'_down as [k2' IHn'_down].
    rewrite -> IHn'_down.
    exists (fib (S n'') + k2').
    rewrite -> Nat.mul_add_distr_l.
    reflexivity.
```

2.4.1 A slight refinement

Notice that we do not make use of $P(n+1)$ or $P(n+2)$ in our proof and the "waste" suggests that a more apt induction principle can be used, and indeed this is the case,

consider the induction principle:

```
Lemma nat_ind33 :
  forall P : nat -> Prop,
    P 0 ->
    P 1 ->
    P 2 ->
    (forall n : nat, P n -> P (S (S (S n)))) ->
    forall n : nat, P n.
```

Now, we can use the exact same proof as before, but we can use a more fitting induction principle. The same change in induction principle may be made for the forwards direction in the proof.

2.4.2 A less slight refinement

Something that is striking to me is that we are inducting upon the entire set of natural numbers, when we just want to do so for the multiples of three.

The author's main gripe with the proof so far is that we are inducting upon the set:

$$0, 1, 2, 3, 4, 5 \cdots$$

When we only want to induct upon the set:

$$0, 3, 6, 9, 12, 15 \cdots$$

We don't care about what happens outside the set for this forwards direction. The rest of the natural numbers contradict our assumption that the number under consideration is a multiple of three, so let's just get rid of them at the outset.

So we have a custom induction principle:

```
Lemma mul3_ind :
  forall P : nat -> Prop,
    P 0 ->
    (forall n : nat,
      P n -> P (S (S (S n)))) ->
      forall n : nat,
        P (3 * n).
```

While the induction principle above is suited for the task, the the induction tactic in tCPA does not work with this induction principle[5].

The solution is to then brute force the induction out using the `ex_intro` tactic.

```
Check (ex_intro
        (fun k2 : nat => fib 0 = 2 * k2)
            0
            wanted_0).
```

So this settles the base case, and then we need the assumption for $P(n)$, which is formally rendered as:

```
    Check (mul3_ind
        (fun n => exists k2 : nat, fib n = 2 * k2)
        (ex_intro
            (fun k2 : nat => fib 0 = 2 * k2)
             0
            wanted_0)).
```

Now, we can apply the aforementioned induction principle and then see how our goal has changed to become:

```
    forall n : nat,
    (exists k2 : nat, fib n = 2 * k2) ->
    exists k2 : nat, fib (S (S (S n))) = 2 * k2
```

[5]For reasons that the author is not entirely sure of due to the time constraints.

Now, we can use the unfolding idea described above.

```
    intros n [k2 H_k2].
    rewrite -> fold_unfold_fib_SSS.
    rewrite -> H_k2.
    rewrite <- Nat.mul_add_distr_l.
    exists (fib (S n) + k2).
    reflexivity.
```

This proof is much simpler than the previous ones, probably because we are hand-rolling our own custom made induction principle. Someone who only how to use weak and strong induction would produce a much worse proof.

2.4.3 Another refinement!

If we look even more carefully, we *really* want to induct upon the set

$$F_0, F_3, F_6, F_9, ...$$

So, we can state the induction principle as:

```
Lemma fib3_ind:
  forall P : nat -> Prop,
    P (fib 0) ->
    (forall n : nat,
        P (fib n) -> P (fib (S (S (S n))))) ->
        forall n : nat,
          P (fib (3 * n)).
```

Now this makes our proof all the more tigher, as we can see clearly that the proof becomes simple and needs no extra lemmas or hypotheses:

```
  assert (wanted_0 :
           exists q : nat, fib 0 = 2 * q).
```

```
  {
    exists 0.
    compute.
    reflexivity.
  }
  apply (fib3_ind
           (fun m : nat => exists q : nat, m = 2 * q)
           wanted_0
         ).
  intros n [q H_q].
  rewrite -> fold_unfold_fib_SSS.
  rewrite -> H_q.
  rewrite <- Nat.mul_add_distr_l.
  exists (fib (S n) + q).
  reflexivity.
```

The next logical step was to formulate the induction hypothesis and to see what would happen for the proof of the forwards direction:

```
Lemma nat2_ind:
  forall P : nat -> Prop,
    P (2 * 0) ->
    (forall n : nat,
        P (2 * n) -> P (S (S (2 * n)))) ->
    forall n : nat,
      P (2 * n).
```

In this case, we got stuck in the proof as we did not have the lemmas to reason about the successor of the successor of Fibonacci numbers.

Joker: "This works for even numbers, what about for odd numbers?"

Jester: "Well for odd numbers, we have:"

```
Lemma natS2_ind:
  forall P : nat -> Prop,
    P (S (2 * 0)) ->
```

```
(forall n : nat,
    P (S (2 * n)) -> P (S (S (S (2 * n))))) ->
forall n : nat,
  P (S (2 * n)).
```

2.4.4 A proof by strong induction

Here is a Proof from the Wastebasket: Using strong induction, we could have sloppily argued to the effect that since $P(n)$ is even, then $P(n-3)$ must have been even, then $P(n-2)$ and $P(n-1)$ must both be odd, because if even one of them is even, then the Fibonacci series must only contain even numbers from that point out, which is absurd.[6]

2.4.5 Why this Proof was thrown out of The Book

Joker: "That is a valid proof that we have above! Why has it been thrown out of The Book and into the wastebasket!"

Jack: "It is indeed valid, but notice that we need the result that two consecutive Fibonacci numbers cannot be even for the contradiction to work."

Prof Dijkstra: "A student of mine, would *not* have written a proof like this. The avoidable use of reductio ad absurdum is a sign of sloppy thinking."[7]

Prof Danvy: "There is also the use of subtraction, which we don't have many lemmas to reason about. You are thinking computationally, when you could be thinking structurally."

Jester: "Well, the proof still works. These are just aesthetic considerations."

Prof Danvy: "Well, you have also happily ignored the condition that n must be strictly greater than 2...."

Jester (flushed): "I suppose that makes sense."

[6] Admittedly, this was the first proof that came to the author's mind when the problem was presented to him in the lecture.

[7] Dijkstra, Edsgar *From "Discrete Mathematics with Applications" by Susanna S. Epp. EWD 1076*

Jack: "Prof Dijkstra, Prof Danvy, thanks for stopping by."

2.5 Second proof of the backwards direction

Another possible proof direction would be to use `nat_ind2` was suggested by Prof Danvy.

This proof is much shorter than the proof that was previously considered. The induction happens on m instead of n, this is a key difference between this proof and the previous one.

So the induction statement is begun using:

```
induction m as [ | | m'' _[q' IHSm'']] using nat_ind2.
```

While the rest of the formal proof was routine and mechanical, it is instructive to consider what happened in plain English.

Using the hypothesis `IHSm'' : fib (3 * S m'') = 2 * q'`, we want to show that `exists k2 : nat, fib (S (S (S (3 * S m'')))) = 2 * k2 .`

The natural thing to do in this case is to unfold the definition using our fold-unfold lemma `fold_unfold_fib_SSS.`

After this was done, we simply had to prove:

```
exists k2 : nat,
    2 * fib (S (3 * S m'')) + fib (3 * S m'') = 2 * k2
```

And it is clear that we can let `k2` be `fib (S (3 * S m'')) + 2 * q'`.

Joker: "What is the difference between this proof and the previous one using `nat_ind3`? Why is the proof with `nat_ind2` so much shorter."

Jack: "Well, notice that for the proof with `nat_ind3`, we were inducting on n and needed to use the `shifting_by_threes` so that we could use our induction hypotheses to conclude

the proof."

Joker: "What if I were to induct on *m* for the `nat_ind3` case?"

Jack: "Well, you still can't avoid using the `shifting_by_threes` lemmas. Your hypotheses would then include `H_n : n = 3 * S (S (S m''))`, while your induction hypotheses will call for the use of `n = 3 * S (S m'')`, `n = 3 * S m''` and `n = 3 * m''` in the antecedent. To get here, the shifting by threes lemmas is needed."

Joker: "Oh, I think I get it. In the `nat_ind2` case, we don't need the extra lemma as we can use the fold-unfold lemmas associated with the Fibonacci function to use our induction hypotheses."

Jack (Happy): "Yes".

2.6 What about the Odd Fibonacci numbers?

Going back to the table in the first section 2.1, we may formulate the conjecture that:

```
Proposition odd_fibonacci_numbers :
    forall n : nat,
      (exists q : nat, fib n = S (2 * q)) <->
      (exists m : nat, n = S (3 * m) \/ n = S (S (3 * m))).
```

Learning from our previous experience, it was natural to use `nat_ind3` for the forwards direction and `nat_ind2` for the backwards direction to get simple proofs in either direction.

The structure of the proofs are largely the same as the ones above, though for the forwards direction, we did need the new lemma:

```
Lemma about_sum_to_odd_numbers:
  forall a b c : nat,
    2*a + b = S(2*c) ->
      exists k : nat,
          b = S(2 * k).
```

This can be proven using some algebraic manipulation and subtraction or with structural induction as we did before.

The proof itself is simple, at times, one needs to know if we are in the left or right case of the disjunction, so the proof is structured as such:

```
     Check (about_sum_to_odd_numbers
               (fib (S n''))
               (fib    n'')
               q
               H_q).
      Check (IHn' (about_sum_to_odd_numbers
                     (fib (S n''))
                     (fib    n'')
```

```
                  q
                  H_q)).
    destruct (IHn' (about_sum_to_odd_numbers
                     (fib (S n''))
                     (fib    n'')
                     q
                     H_q))
      as [m [H_m | H_m]].

  - rewrite -> H_m.
      rewrite -> thrice_S.
      exists (S m).
      left.
      reflexivity.

    - rewrite -> H_m.
      rewrite -> thrice_S.
      exists (S m).
      right.
      reflexivity.
```

Informally, the left case considers when `n'' = S (3 * m)` and the right case considers when `n'' = S (S (3 * m))`. So what we have here is essentially a proof done by splitting into cases.

The situation is similar in the backwards case, where we proceed by `nat_ind2`

One could also proceed by `nat_ind3`, but this would require us to usethe `shifting_by_threes` lemma.

The base cases are trivial, and for the induction hypotheses one needs to consider the left and right cases seperately. So for the left case, we need to show that:

"`exists q : nat, S (2 * (S (2 * q') + fib (3 * S m'') + q')) = S (2 * q)`".

While for the right case, we need to show that:

"`exists q : nat, S (2 * (S (2 * q') + fib (3 * S m'') + q')) = S (2 * q)`".

In plain English, here the left case is when we are dealing with the n-th Fibonacci term where n has a remainder of 1 when divided by 3 while the rght case is when the remainder is 2.

It is then clear what value of `q` to introduce to conclude this proof.

2.7 Lagniappe: An artful take on this problem

This problem is discussed lightly in Donald Knuth's *The Art of Computer Programming*. If we turn to page 81 of section 1.2.8 in *Volume 1: Fundamental Algorithms* which treats the Fibonacci numbers, we see that:

"From the definition we find immediately that

$$F_{n+3} = F_{n+2} + F_{n+1} = 2F_{n+1} + F_n;\ F_{n+4} = 3F_{n+1} + 2F_n$$

and, in general, by induction that

$$F_{n+m} = F_m F_{n+1} + F_{m+1} F_n$$

for any positive integer m.

If we take m to be a multiple of n [above], we find inductively that

$$F_{nk} \text{ is a multiple of } F_n.$$

Thus every third number is even, every fourth number is a multiple of 3, every fifth is a multiple of 5, and so on."

However, it should be noted that at this stage the book does not explicitly show how if a Fibonacci number is even, it is a multiple of three.

Still it is interesting to formalise this argument in tCPA. First we formalise the addition argument by doing the following:

```
    Lemma knuth_add:
  forall n m : nat,
    fib (n + (S m)) = fib (S m) * fib (S n) + fib m * fib n.

Proof.
  induction n using nat_ind2; intros m.

  * rewrite -> Nat.add_0_l.
    rewrite -> fold_unfold_fib_S.
    rewrite -> Nat.mul_1_r.
    rewrite -> fold_unfold_fib_O.
    rewrite -> Nat.mul_0_r.
    rewrite -> Nat.add_0_r.
    reflexivity.

  * rewrite ->fold_unfold_fib_S.
    simpl (fib 2).
    rewrite ->2 Nat.mul_1_r.
    reflexivity.

  * assert (H : forall n : nat,
              S (S n) + S m = S (S (n + S m))).
    {
      intros j.
      ring.
    }
    rewrite -> H.
    clear H.
    rewrite -> fold_unfold_fib_S.
    rewrite -> IHn.
    assert (H: forall n : nat,
              S (n + (S m)) = n + (S (S m))).
    {
      intros j.
      ring.
    }
    rewrite -> H.
    rewrite -> IHn.
    rewrite -> fold_unfold_fib_S.
    rewrite -> (fold_unfold_fib_S (S (S n))).
```

```
    rewrite -> (fold_unfold_fib_S (S n)).
    ring.
Qed.
```

This gives rise to the proof below:

```
Lemma knuth_times:
  forall n k : nat,
    exists m : nat,
    fib ((S n) * (S k)) = m * fib (S n).

Proof.
  intros n k.
  revert n.
  induction k as [ | n' IHn']; intros n.

  * exists 1.
    rewrite -> Nat.mul_1_r.
    rewrite -> Nat.mul_1_l.
    reflexivity.

  * assert (H: forall n k : nat,
                (S n * (S (S k)) =
                (S n) * (S k) + (S n))).
    {
      intros a b.
      ring.
    }
    rewrite -> H.
    rewrite -> knuth_add.
    destruct (IHn' n) as [m IHm].
    rewrite -> IHm.
    exists (fib (S (S n * S n')) + fib n * m).
    ring.
Qed.
```

Notice that we induct upon k and not upon n.

The author of this report did not originally pay such careful attention to the corner cases when reading the corresponding section by Knuth. See how n and k must be positive integers?

This is critical and important to pay attention to, but the author had completely missed it when first trying to formalise the proof.

It is sobering to see how aware formal proofs make us.

Knuth then goes on to state and prove the more general theorem of Lucas using the Euclidean Algorithm, which says that:

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}$$

If one takes this theorem on faith, the problem we are dealing with becomes much easier and a reader wishing to investigate generalisations of the problem at hand may find this theorem helpful.

2.8 Vantage Loaf: Proof Sketch with Modular Arithmetic

If the results of modular arithmetic are available to us, this problem becomes much easier. Strong induction will be used for this proof sketch, but it may be naturally refined to use `nat_ind3` if the reader wishes to formalise the below argument in tCPA.

First note that $F_0 \pmod 2 = 0$, $F_1 \pmod 2 = 1$ and $F_3 \pmod 2 = 1$.

Next observe that:

$$F_{n+3} \bmod 2 = (2F_{n+1} \bmod 2 + F_n \bmod 2) \bmod 2 = (F_n \bmod 2) \bmod 2 = F_n \bmod 2$$

Then, by induction, it is almost immediately clear that if a number is divisible by three, then by induction it's corresponding Fibonacci number is even.

3 Polygonal Numbers

3.1 Identifying the theorem

Prof Danvy presents the the following result in his paper *Summa Summarum*.

In plain English he tells us that "for any given positive integer k and for all natural numbers n,

$$\sum_{i=0}^{k\cdot(n+1)} \left\lfloor \frac{i}{k} \right\rfloor$$

is a $(k+2)$-th polygonal number."

Which is rendered as:

$$\sum_{i=0}^{k\cdot(n+1)} \left\lfloor \frac{i}{k} \right\rfloor = \sum_{i=0}^{n} (k \cdot i + 1) = k \cdot \sum_{i=0}^{n} i + n + 1$$

It is a classic result in elementary number theory[8] that the general formula for the k-th polygonal number is:

$$P_k(n) = \frac{(k-2)n^2 - (k-4)n}{2},$$

where k is the number of sides of the polygon and n is the n-th term.

So, we essentially have the result that:

$$\sum_{i=0}^{k\cdot(n+1)} \left\lfloor \frac{i}{k} \right\rfloor = \frac{kn^2 - (k-2)n}{2}$$

Let us verify this lemma for two cases:

[8]Weisstein, Eric W. "Polygonal Numbers". MathWorld.

3.2 Squares and Floors

First, if we were to look at when $k = 2$, we have that:

$$\forall n \in \mathbb{N}, \quad \sum_{i=0}^{2 \cdot n} \left\lfloor \frac{i}{2} \right\rfloor = n^2$$

Formally, this is rendered in tCPA as such:

```
Lemma square_floors:
  forall n : nat,
    Sigma (2*n) (fun i : nat => i / 2) = n*n.
```

The proof is done by induction as per usual for the base cases:

```
  intros n.
  induction n as [ | n' IHn'].

  * rewrite -> Nat.mul_0_r.
    rewrite -> fold_unfold_Sigma_0.
    simpl.
    reflexivity.

  * rewrite <- twice_S.
    rewrite ->2 fold_unfold_Sigma_S.
    rewrite -> IHn'.
```

The induction step requires us to show that:

```
Sigma (S (S (2 * n'))) (fun i : nat => i / 2) = S n' * S n
```

Using the fold-unfold lemmas and the induction hypothesis, we have that:

```
    n' * n' + S (2 * n') / 2 + (1 + n') * 2 / 2 = S n' * S n'
```

The author was mildly panicked when he saw that he had to show that essentially:

$n^2 + 2n + 1 + \frac{1}{2} = n^2 + 2n + 1$

But heaved a sigh of relief when it was clear that $\frac{1}{2}$ is 0 since we are working with division in the natural numbers.

Once this insight is clear, it is a matter of coaxing out the proof using the appropriate tactics for integer division in tCPA as done below:

```
rewrite <- twice_S.
rewrite ->2 fold_unfold_Sigma_S.
rewrite -> IHn'.

assert (H: forall n : nat, S (S (2 * n)) = (1 + n)*2).
{
  intros n.
  ring.
}
rewrite -> H.
Search (_ * ( _) / _).
rewrite -> (Nat.div_mul (1 + n') 2 (Nat.neq_succ_0 1)).

Search ((_ + _) / _ ).
rewrite <- Nat.add_1_l.
rewrite -> (Nat.mul_comm 2 n').
Search ((_ + _) / _ ).
rewrite -> (Nat.div_add 1 n' 2 (Nat.neq_succ_0 1)).
simpl (1/2).
ring.
```

It should be noted that this is different from the proof in the aforementioned paper, since that proof manipulates the lower bound of the sum, while we have no provision to do so, and so did not do so.

3.3 On an exercise by Knuth

Exercise 1.2.6-36 in *The Art of Computer Programming*, asks the reader to prove that:

$$\sum_{k=1}^{n} \left\lfloor \frac{k}{2} \right\rfloor = \left\lfloor \frac{n^2}{4} \right\rfloor.$$

This sum is slightly harder to evaluate, since it requires us to distinguish between the cases when n is even and when n is odd.

But with this, the previous result by Prof Danvy may be established by setting the upper bound to be $2n$ and observing that as a result:

$$\lfloor \frac{4n^2}{4} \rfloor = \lfloor n^2 \rfloor$$

Since n is an integer and integer multiplication is closed under the set of natural numbers, we have that $\lfloor n^2 \rfloor = n^2$.

So, we may readily conclude that:

$$\sum_{k=1}^{2n} \left\lfloor \frac{k}{2} \right\rfloor = n^2.$$

However, one can also go in the opposite direction, using the result that $\sum_{k=1}^{n} \lfloor \frac{k}{2} \rfloor = \lfloor \frac{n^2}{4} \rfloor$, one can substitute in $m = \frac{n}{2}$ and then do a case analysis to solve the exercise.

For example, we may see that:

```
Lemma knuth:
  forall n : nat,
    Sigma n (fun i : nat => i / 2) = n*n / 4.

Proof.
  intros n.
  induction n as [ | n' IHn'].
```

```
* compute.
  reflexivity.

* destruct (odd_or_even n') as [m' [H_e | H_o]].

  + rewrite -> H_e.
    rewrite -> fold_unfold_Sigma_S.
    rewrite -> (square_floors m').
    ...

  + rewrite -> H_o.
    rewrite -> twice_S.
    rewrite -> (square_floors (S m')).
    ...
```

The omitted parts are algebraic manipulations with the division symbol. During these manipulations, it is typical to encounter typical cases where one needs to prove expressions where the left hand side and the right hand side appear to not be equal. In such cases, as long as one remembers that fractional terms like $\frac{m}{n}$ go to zero as long as m and n which are natural numbers with $m > n$, there will be no issue.

For example, the even case above is treated as such:

```
    rewrite -> H_e.
    rewrite -> fold_unfold_Sigma_S.
    rewrite -> (square_floors m').
    rewrite <- Nat.add_1_r.
    assert (H: forall n : nat,
               (1 + 2 * m') / 2 = m').
    {
      intros n.
      Search ((_ + _*_) / _).
      rewrite -> Nat.mul_comm.
      rewrite -> (Nat.div_add 1 m' 2 (Nat.neq_succ_0 1)).
      simpl (1/2).
      reflexivity.
    }
    rewrite -> (Nat.add_comm (2 * m') 1).
```

```
      rewrite -> (H m').
      clear H.
      assert (H : forall n : nat,
                  (1 + 2 * n) * (1 + 2 * n) =
                  1 + (n*n + n)*4).
      {
        intros n.
        reflexivity.
      }
      rewrite -> (H m').
      Check Nat.div_add.
      rewrite -> (Nat.div_add _ _ 4 (Nat.neq_succ_0 3)).
      simpl (1/4).
      ring.
}
```

So that's part of an *M23*-rated exercise done in tCPA!

3.4 A Natural (Number) Deconstruction of Prof Danvy's Curiosa.

Eventually, expelling rabbits became another joy of my professional life.

— Edsgar W Dijkstra
EWD1300

Next, we consider an intuitive explaination for why the floor formula for polygonal numbers works, only using elementary mathematics.

Andrei Kolomogrov discovered that the sum of the first n odd numbers was the n-th square number. First, we begin by writing down the result he published in his school journal *The Swallow of Spring*:

$$1 + 3 + 5 + 7 + 9 + \cdots + 2n + 1 = n^2$$

The next thing to note is that if we have an odd number m, then it can be written as

$$\frac{(m-1)}{2} + \frac{(m-1)}{2} + 1 = \frac{(m-1)}{2} + (\frac{(m-1)}{2} + 1)$$

Expanding and rearranging the terms, we have,

$$0 + 1 + 1 + 2 + 2 + 3 + 3 + 4 + 4 + 5 + 5 \cdots$$

And it just so happens to be the case that the sum of the first $2n$ terms corresponds to the n-th triangular number. This is how the $2n$ appears.

Now, the next step is to see what happens when we make three copies of one (The mechanism to control the number of copies we make is the floor function. If we divide by n, we make n copies. If we multiply by n, we eliminate all the copies between 1 and n).

$$0 + 0 + 1 + 1 + 1 + 2 + 2 + 2 + 3 + 3 + 3 + 4 + 4 + 4 + 5 + 5 + 5...$$

Before, we had to group two by two, now we will group three by three.

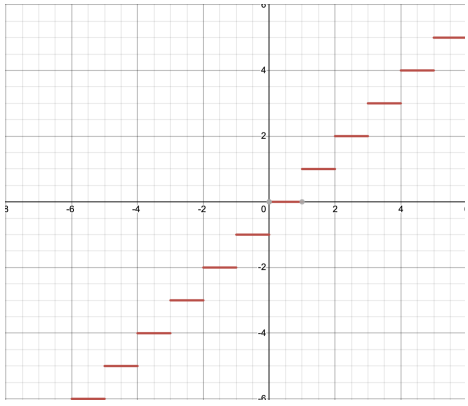
$$(0+0+1)+(1+1+2)+(2+2+3)+(3+3+4)+(5+5+6)+\cdots$$

Now, this is the sum $3n+1$ from 1 to n. Similarly, we get the sum $4n+1$ from 1 to n which generates the hexagonal numbers and so on. Now by the floor function, if you want k copies of numbers from 1 to n, you divide by k and run it from 1 to kn.

3.5 Some geometric intuition

It was said without any reasoning that the mechanism to control the number of copies we make is the floor function.

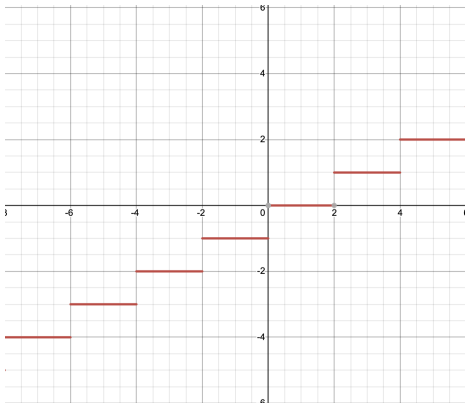
Here is the floor function $y = \lfloor x \rfloor$:



Now this begs us to investigate what happens when we scale the graph by a factor of 2

Here is the scaled floor function $y = \lfloor \frac{x}{2} \rfloor$:

This is a classic result coordinate geometry: When we perform the graph transformation $f(x) \to f(ax)$, the graph is scaled horizontally by a factor of $\frac{1}{a}$.



3.6 Lagniappe: Some additional series computed with floors

Now some series are constructed using the floor function without proof. The purpose is to demonstrate how much we understand the scaling property.

To warm up, let us consider:

$\sum_{k^2}^{(k^2+k-1)} \lfloor \frac{i}{k} \rfloor = k^2$

and also:

$\sum_{k(k-1)}^{k^2} \lfloor \frac{i}{k} \rfloor = k^2$

Things get a lot more interesting if we try to *nest* these floors, consider the below pattern:

1
11
111
1111
...

This can be expressed as the below nested sum:

$\sum_{k=1}^{n} \sum_{j=k}^{2k-1} \lfloor (j/k) \rfloor$

3.6.1 A conjecture on factorials

Consider the factorials. Suppose we already have $n!$ we can ask, how we can we use this to go to $(n+1)!$?

The answer is that we need to make n copies of $(n+1)!$.

Now the experimental guess below came after first computing and realising that:

$\sum_{120(7)}^{120(7)+5} \lfloor (j/7) \rfloor = 6!$
$\sum_{720(8)}^{720(7)+7} \lfloor (j/8) \rfloor = 7!$
$\sum_{5040(9)}^{5040(9)+7} \lfloor (j/9) \rfloor = 8!$

So it is reasonable to conjecture that:

For all $n \in \mathbb{N}, n > 2$:

$$\sum_{(n-2)!n}^{(n-2)!(n)+n-2} \lfloor (j/n) \rfloor = (n-1)!$$

The conjecture can be refined by substituting n for $n+2$ so that the hypothesis can be eliminated.

For all $n \in \mathbb{N}$:

$$\sum_{(n)!(n+2)}^{(n!(n+2)+n} \lfloor (j/(n+2)) \rfloor = (n+1)!$$

The intuition for why this works is based on the idea that we are taking $n!$, making $n+1$ copies of this and then we add these copies up. But the proof of this is outside the scope of this project.

3.6.2 A conjecture on Möbius polynomials

While experimenting with the series, it was natural to try to guess a nested series of the form:

$\sum_{i=1}^{n} \sum_{j=1}^{i} \lfloor \frac{i}{j} \rfloor$

We may conjecture that the the difference between the successive outputs of this function correspond to the coefficient of x in the n-th Möbius polynomial after plugging in the terms into the Online Encyclopaedia of Integer Sequences.[9]

[9]Murray, Will. "Möbius polynomials." Mathematics Magazine 85, no. 5 (2012): 376-383.

4 Conclusion

So we have looked at formal proofs of two problems in elementary number theory.

Some may feel that this project is not substaintial because spends lots of time looking a theorem which can be solved in one or two lines using modular arithmetic and Lucas's theorem. It may seem that spending lots of time finding simpler and more elementary proofs of a trivial result has no use in us moving on to prove newer and more novel results.

But notice how with the tigher induction principles, our proof becomes far more clearer. Concretely there are far lesser unnecessary cases which we need to consider and we use fewer hypotheses, which makes for a simpler proof.

By spending so much time on these seemingly innocuous and simple questions, we have managed to find links between these questions and those already asked in the literature that were not seen thus far (for example, by establishing the connection between squares and the exercise by Knuth).

4.1 A Direct Communication

I now speak directly to the reader of the report. While I could come up with more technical reasons to justify this project, I think there is inherent aesthetic value in finding clear and tight proofs of known results in mathematics.

But also, as Dijkstra once said *"Elegance is not a dispensable luxury, but a must, a "matter of life and death" so to speak...The reason why, so often, mathematicians can come away with a rather inelegant way of working is probably that, at least in comparison to software projects, mathematical projects are relatively "small". Yet I regret the general lack of elegance: it makes mathematical texts less attractive to write and harder to enjoy."*

I hope that this kind of project, where the goal is not to rush after new problems and results, but to clarify what was already known in the Ehrenfrest tradition, is carried out in the future to help clarify and simplify topics in mathematics for everyone's enjoyment.

And so I would like to conclude this project with another quote by Dijkstra.

"The applied mathematicians that solved the differential equations should, to my taste, have realized that they were cracking an egg with a sledgehammer –an activity I don't like, no matter how effective a sledgehammer may be for cracking eggs-"