Formulations and solutions of IMO problems in Isabelle/HOL

Filip Marić and Sana Stojanović-Đurđević

June 12, 2020

Contents

1 IMO 2006 SL statements **5**
- 1.1 Algebra problems . . . 5
  - 1.1.1 IMO 2006 SL - A1 . . . 5
  - 1.1.2 IMO 2006 SL - A2 . . . 6
  - 1.1.3 IMO 2006 SL - A3 . . . 6
  - 1.1.4 IMO 2006 SL - A4 . . . 6
  - 1.1.5 IMO 2006 SL - A5 . . . 7
  - 1.1.6 IMO 2006 SL - A6 . . . 7

2 IMO 2008 SL statements **9**
- 2.1 Algebra problems . . . 9
  - 2.1.1 IMO 2008 SL - A1 . . . 9

3 IMO 2018 SL statements **11**
- 3.1 Algebra problems . . . 11
  - 3.1.1 IMO 2018 SL - A1 . . . 11
  - 3.1.2 IMO 2018 SL - A2 . . . 12
  - 3.1.3 IMO 2018 SL - A3 . . . 12
  - 3.1.4 IMO 2018 SL - A4 . . . 12
  - 3.1.5 IMO 2018 SL - A5 . . . 13
  - 3.1.6 IMO 2018 SL - A7 . . . 13

4 IMO 2006 SL solutions **15**
- 4.1 Algebra problems . . . 15
  - 4.1.1 IMO 2006 SL - A2 . . . 15

5 IMO 2018 SL solutions **19**
- 5.1 Algebra problems . . . 19

5.1.1 IMO 2018 SL - A2 19
5.1.2 IMO 2018 SL - A4 29
5.2 Combinatorics problems 47
5.2.1 IMO 2018 SL - C1 47
5.2.2 IMO 2018 SL - C2 59
5.2.3 IMO 2018 SL - C3 83
5.2.4 IMO 2018 SL - C4 138
5.3 Number theory problems 166
5.3.1 IMO 2018 SL - N5 166

Chapter 1

IMO 2006 SL statements

theory *IMO-2006-SL-statements*
imports *Main*

begin

Shortlisted problems with solutions from *47-th International Mathematical Olympiad, 2006, Slovenia.*

File with problem statements and solutions can be found at: https://www.imo-official.org/problems/IMO2006SL.pdf

end

1.1 Algebra problems

1.1.1 IMO 2006 SL - A1

theory *IMO-2006-SL-A1*
imports *Complex-Main*
begin

theorem *IMO-2006-SL-A1*:
fixes $a :: nat \Rightarrow real$
assumes $\forall\ i \geq 0.\ (a\ (i + 1) = floor\ (a\ i) * (a\ i - floor\ (a\ i)))$
shows $\exists\ j.\ a\ j = a\ (j + 2)$
sorry

end

1.1.2 IMO 2006 SL - A2

theory *IMO-2006-SL-A2*
imports *Complex-Main*
begin

theorem *IMO-2006-SL-A2*:
fixes $a :: nat \Rightarrow real$
assumes $a\ 0 = -1\ \forall\ n \geq 1.\ (\sum\ k < Suc\ n.\ a\ (n - k)\ /\ (k + 1)) = 0\ n \geq 1$
shows $a\ n > 0$
using ‹$n \geq 1$›
sorry

end

1.1.3 IMO 2006 SL - A3

theory *IMO-2006-SL-A3*
imports *Complex-Main Tutorial.Pairs*

begin

theorem *IMO-2006-SL-A3*:
fixes $c :: nat \Rightarrow nat$
and $S :: (nat \times nat)\ set$
assumes $c\ 0 = 1\ c\ 1 = 0\ \forall\ n \geq 0.\ (c\ (n + 2) = c\ (n + 1) + c\ n)$ **and**
$\forall\ (x,y) \in S.\ \exists\ J :: nat\ set.\ (\forall\ j \in J.\ j > 0) \wedge$
$(\sum j \in J.\ c\ j) = x \wedge (\sum j \in J.\ c\ (j-1)) = y$
shows $\exists\ \alpha\ \beta\ m\ M ::$
$real.\ (x,y) \in S \longleftrightarrow (m < \alpha * x + \beta * y \wedge\ \alpha * x + \beta * y\ < M)$
sorry

end

1.1.4 IMO 2006 SL - A4

theory *IMO-2006-SL-A4*
imports *Complex-Main*
begin

theorem *IMO-2006-SL-A4*:

fixes $a :: nat \Rightarrow real$ **and** $n :: nat$
assumes
$\forall\ i.\ 1 \leq i \wedge i \leq n \wedge a\ i > 0$
$\forall\ n \geq 1.\ (\sum\ k < Suc\ n.\ a\ (n - k)\ /\ (k + 1)) = 0\ n \geq 1$
shows
$(\sum\ i \leftarrow [1..<n].\ \sum\ j \leftarrow [i+1..<n+1].\ (a\ i * a\ j\ /\ (a\ i + a\ j))) \leq$
$(n\ /\ (2 * (\sum\ i \leftarrow [1..<n+1].\ a\ i)) * (\sum\ i \leftarrow [1..<n].\ \sum\ j \leftarrow [i+1..<n+1].\ (a\ i * a\ j)))$
sorry

end

1.1.5 IMO 2006 SL - A5

theory *IMO-2006-SL-A5*
imports *Complex-Main HOL−ex.Sqrt*
begin

theorem *IMO-2006-SL-A5*:
fixes $a\ b\ c :: nat$
assumes $a > 0\ b > 0\ c > 0\ a + b > c\ b + c > a\ c + a > b$
shows $sqrt\ (b + c - a)\ /\ (sqrt\ b + sqrt\ c - sqrt\ a) +$
$sqrt\ (c + a - b)\ /\ (sqrt\ c + sqrt\ a - sqrt\ b) +$
$sqrt\ (a + b - c)\ /\ (sqrt\ a + sqrt\ b - sqrt\ c) \leq 3$
sorry

end

1.1.6 IMO 2006 SL - A6

theory *IMO-2006-SL-A6*
imports *Complex-Main HOL−ex.Sqrt*
begin

theorem *IMO-2006-SL-A6*:
fixes $a\ b\ c :: real$
shows $Max\ \{(a * b * (a * a - b * b) + b * c * (b * b - c * c) + c * a * (c * c - a * a))\ /\ (a * a + b * b + c * c)\} = (sqrt\ 2) * 9\ /\ 32$
sorry

end

Chapter 2

IMO 2008 SL statements

theory *IMO-2008-SL-statements*
imports *Main*

begin

Shortlisted problems with solutions from *49-th International Mathematical Olympiad, July 10-22, 2008, Madrid, Spain.*

File with problem statements and solutions can be found at: https://www.imo-official.org/problems/IMO2008SL.pdf

end

2.1 Algebra problems

2.1.1 IMO 2008 SL - A1

theory *IMO-2008-SL-A1*
imports *Complex-Main*
begin

theorem *IMO-2008-SL-A1*:
fixes f :: $real \Rightarrow real$
assumes $\forall\ p\ q\ r\ s$ :: $real.\ p > 0 \wedge q > 0 \wedge r > 0 \wedge s > 0 \wedge pq = rs \wedge$
$((f\ p)\hat{}2 + (f\ q)\hat{}2)\ /\ ((f\ r\hat{}2) + (f\ s\hat{}2)) = (p\hat{}2 + q\hat{}2)\ /\ (r\hat{}2 + s\hat{}2)$
shows $(\forall\ x > 0.\ f\ x = x) \vee (\forall\ x > 0.\ f\ x = 1\ /\ x)$
sorry

end

Chapter 3

IMO 2018 SL statements

theory *IMO-2018-SL-statements*
imports *Main*

begin

Shortlisted problems with solutions from *59-th International Mathematical Olympiad, 3-14 July 2018, Cluj-Napoca, Romania.*

File with problem statements and solutions can be found at: https://www.imo-official.org/problems/IMO2018SL.pdf

end

3.1 Algebra problems

3.1.1 IMO 2018 SL - A1

theory *IMO-2018-SL-A1*
imports *HOL.Rat*

begin

theorem *IMO2018SL-A1*:
fixes $x\ y :: rat$ **and** $f :: rat \Rightarrow rat$
assumes $f\ (x * x * (f\ y) * (f\ y)) = (f\ x) * (f\ x) * (f\ y)$
shows $f\ x = 1$
sorry

end

3.1.2 IMO 2018 SL - A2

theory *IMO-2018-SL-A2*
imports *Complex-Main*
begin

theorem *IMO2018SL-A2*:
fixes $n :: nat$
assumes $n \geq 3$
shows $(\exists\ a :: nat \Rightarrow real.\ a\ n = a\ 0 \wedge a\ (n+1) = a\ 1 \wedge$
$(\forall\ i < n.\ (a\ i) * (a\ (i+1)) + 1 = a\ (i+2))) \longleftrightarrow$
$3\ dvd\ n$ (**is** $(\exists\ a.\ ?p1\ a \wedge ?p2\ a \wedge ?eq\ a) \longleftrightarrow 3\ dvd\ n$)
sorry

end

3.1.3 IMO 2018 SL - A3

theory *IMO-2018-SL-A3*
imports *Complex-Main*

begin

theorem *IMO2018SL-A3*:
fixes $S :: nat\ set$
assumes $\forall\ x \in S.\ x > 0$
shows $(\exists\ F\ G.\ F \subseteq S \wedge G \subseteq S \wedge F \cap G = \{\} \wedge (\sum x{\in}F.\ 1/(rat\text{-}of\text{-}nat\ x))$
$= (\sum x{\in}G.1/(rat\text{-}of\text{-}nat\ x))) \vee$
$(\exists\ r{::}rat.\ 0 < r \wedge r < 1 \wedge (\forall\ F \subseteq S.\ finite\ F \longrightarrow (\sum x{\in}F.\ 1/(rat\text{-}of\text{-}nat$
$x)) \neq r))$
sorry

end

3.1.4 IMO 2018 SL - A4

theory *IMO-2018-SL-A4*
imports *Complex-Main*

begin

definition *is-Max* :: *'a::linorder set* ⇒ *'a* ⇒ *bool* **where**
is-Max A x ⟷ *x* ∈ *A* ∧ (∀ *x'* ∈ *A*. *x'* ≤ *x*)

theorem *IMO2018SL-A4*:
shows
is-Max {*a 2018* − *a 2017* | *a::nat* ⇒ *real*. *a 0* = *0* ∧ *a 1* = *1* ∧ (∀ *n* ≥ *2*. ∃ *k*. *1* ≤ *k* ∧ *k* ≤ *n* ∧ *a n* = (∑ *i* ← [*n*−*k*..<*n*]. *a i*) / *real k*)}
(*2016* / *2017^2*) (**is** *is-Max* {*?f a* | *a*. *?P a*} *?m*)
unfolding *is-Max-def*
sorry

end

3.1.5 IMO 2018 SL - A5

theory *IMO-2018-SL-A5*
imports *Complex-Main*
begin

theorem *IMO-2018-SL-A5*:
fixes *f* :: *real* ⇒ *real*
assumes ∀ *x* > *0*. ∀ *y* > *0*. (*x* + *1*/*x*) ∗ (*f y*) = *f* (*x*∗*y*) + *f* (*y* / *x*)
shows ∃ *C1 C2*. ∀ *x* > *0*. *f x* = *C1* ∗ *x* + *C2* / *x*
sorry

end

3.1.6 IMO 2018 SL - A7

theory *IMO-2018-SL-A7*
imports *Complex-Main*
begin

theorem
shows *Max* {*root 3* (*a* / (*b* + *7*)) + *root 3* (*b* / (*c* + *7*)) + *root 3* (*c* / (*d* + *7*)) + *root 3* (*d* / (*a* + *7*))
| *a b c d* :: *real* . *a* ≥ *0* ∧ *b* ≥ *0* ∧ *c* ≥ *0* ∧ *d* ≥ *0* ∧ *a* + *b* + *c* + *d* = *100*} = *8* / *root 3 7*

sorry

end

Chapter 4

IMO 2006 SL solutions

theory *IMO-2006-SL-solutions*
imports *Main*

begin

Shortlisted problems with solutions from *57-th International Mathematical Olympiad, Slovenia, 2006.*

File with problem statements and solutions can be found at: https://www.imo-official.org/problems/IMO2006SL.pdf

end

4.1 Algebra problems

4.1.1 IMO 2006 SL - A2

theory *IMO-2006-SL-A2-sol*
imports *Complex-Main*
begin

declare [[*smt-timeout* = *20*]]

lemma *sum-remove-zero*:
fixes n :: *nat*
assumes $n > 0$
shows $(\sum\ k < n.\ f\ k) = f\ 0 + (\sum\ k \in \{1..<n\}.\ f\ k)$
using *assms*

```
  by (simp add: atLeast1-lessThan-eq-remove0 sum.remove)

theorem IMO-2006-SL-A2:
  fixes a :: nat ⇒ real
  assumes a 0 = −1 ∀ n ≥ 1. (∑ k < Suc n. a (n − k) / (k + 1)) = 0 n ≥ 1
  shows a n > 0
  using ⟨n ≥ 1⟩
proof (induction n rule: less-induct)
  case (less n)
  show ?case
  proof cases
    assume n = 1
    have a 1 = 1/2
      using assms
      by auto
    with ⟨n = 1⟩ show ?thesis
      by simp
  next
    assume n ≠ 1
    with ⟨n ≥ 1⟩ have n > 1
      by simp

    have 0 = (n + 1) * (∑ k < n + 1. a k / (n + 1 − k)) − n * (∑ k < n. a
k / (n − k))
    proof−
      have (∑ k < n. a k / (n − k)) = 0
        using assms(2)[rule-format, of n − 1] ⟨n > 1⟩
              sum.nat-diff-reindex[of λ k. a k / (n − k) n]
        by simp

      moreover

      have (∑ k < n + 1. a k / (n + 1 − k)) = 0
        using assms(2)[rule-format, of n] ⟨n > 1⟩
              sum.nat-diff-reindex[of λ k. a k / (n + 1 − k) n + 1]
        by simp

      ultimately
      show ?thesis
        by simp
```

```
  qed
  then have (n + 1) * a n = - (∑ k < n. ((n + 1) / (n + 1 - k) - n / (n
- k)) * a k)
    by (simp add: algebra-simps sum-distrib-left sum-subtractf)
  then have (n + 1) * a n = (∑ k < n. (n / (n - k) - (n + 1) / (n + 1 -
k)) * a k)
    by (simp add: algebra-simps sum-negf[symmetric])
  also have ... = (∑ k ∈ {1..<n}. (n / (n - k) - (n + 1) / (n + 1 - k)) *
a k)
    using ‹n > 1›
    by (subst sum-remove-zero, auto)
  also have ... > 0
  proof (rule sum-pos)
    show finite {1..<n}
      by simp
  next
    show {1..<n} ≠ {}
      using ‹n > 1›
      by simp
  next
    fix i
    assume i ∈ {1..<n}
    show (n / (n - i) - (n + 1) / (n + 1 - i)) * a i > 0 (is ?c * a i > 0)
    proof-
      have a i > 0 using less ‹i ∈ {1..<n}› by simp

      moreover have ?c > 0
      proof-
        have ?c = i / ((n - i) * (n + 1 - i))
          using ‹i ∈ {1..<n}›
          by (simp add: field-simps of-nat-diff)
        then show ?thesis
          using ‹i ∈ {1..<n}›
          by simp
      qed

      ultimately show ?thesis by simp
    qed
  qed
  finally have (n + 1) * a n > 0
```

```
    .
  then show ?thesis
    by (smt mult-nonneg-nonpos of-nat-0-le-iff)
 qed
qed

end
```

Chapter 5

IMO 2018 SL solutions

theory *IMO-2018-SL-solutions*
imports *Main*

begin

Shortlisted problems with solutions from *59-th International Mathematical Olympiad, 3-14 July 2018, Cluj-Napoca, Romania.*
File with problem statements and solutions can be found at: https://www.imo-official.org/problems/IMO2018SL.pdf

end

5.1 Algebra problems

5.1.1 IMO 2018 SL - A2

theory *IMO-2018-SL-A2-sol*
imports *Complex-Main*
begin

lemma *n-plus-1-mod-n*:
fixes n :: *nat*
assumes $n > 1$
shows $(n + 1)$ *mod* $n = 1$
by (*metis assms mod-add-self1 mod-less*)

lemma *n-plus-2-mod-n*:

```
  fixes n :: nat
  assumes n > 2
  shows (n + 2) mod n = 2
  by (metis assms mod-add-self1 mod-less)

theorem IMO2018SL-A2:
  fixes n :: nat
  assumes n ≥ 3
  shows (∃ a :: nat ⇒ real. a n = a 0 ∧ a (n+1) = a 1 ∧
                          (∀ i < n. (a i) * (a (i+1)) + 1 = a (i+2))) ⟷
        3 dvd n (is (∃ a. ?p1 a ∧ ?p2 a ∧ ?eq a) ⟷ 3 dvd n)
proof
  assume 3 dvd n

  let ?a = (λ n. if n mod 3 = 0 then 2 else −1) :: nat ⇒ real
  show ∃ a. ?p1 a ∧ ?p2 a ∧ ?eq a
  proof (rule-tac x=?a in exI, safe)
    show ?p1 ?a
      using ⟨3 dvd n⟩
      by auto
  next
    show ?p2 ?a
      using ⟨3 dvd n⟩
      by auto
  next
    fix i
    assume i < n
    show (?a i) * (?a (i+1)) + 1 = ?a (i+2)
      by auto presburger+
  qed
next
  assume ∃ a. ?p1 a ∧ ?p2 a ∧ ?eq a
  then obtain a where ?p1 a ?p2 a ?eq a
    by auto


  let ?a = λ i. a (i mod n)
  have ?p1 ?a ?p2 ?a
    using ⟨?p1 a⟩ ⟨n ≥ 3⟩ n-plus-1-mod-n n-plus-2-mod-n
    by auto
```

```
have eq: ∀ i. ?a i * ?a (i + 1) + 1 = ?a (i + 2)
proof safe
  fix i
  have a ((i + 1) mod n) = a (i mod n + 1)
    using ‹?p1 a›
    by (simp add: mod-Suc)

  moreover

  have a ((i + 2) mod n) = a (i mod n + 2)
    using ‹?p1 a› ‹?p2 a›
    by (metis One-nat-def Suc-eq-plus1 add-Suc-right mod-Suc one-add-one)

  ultimately

  show a (i mod n) * a ((i + 1) mod n) + 1 = a ((i + 2) mod n)
    using ‹?eq a›
    using assms
    by auto
qed

have *: ∀ i. ?a i > 0 ∧ ?a (i + 1) > 0 ⟶ ?a (i + 2) > 1
  using eq
  by (smt mult-pos-pos)

have no-pos-pos: ∀ i. ¬ (?a i > 0 ∧ ?a (i + 1) > 0)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain i where ?a i > 0 ?a (i + 1) > 0
    by auto

  have ∀ j ≥ i+1. ?a j > 0 ∧ ?a (j + 1) > 1
  proof (rule allI, rule impI)
    fix j
    assume i + 1 ≤ j
    then show 0 < ?a j ∧ 1 < ?a (j + 1)
    proof (induction j)
      case 0
      then show ?case
```

```
      by simp
  next
    case (Suc j)
    show ?case
    proof (cases i + 1 ≤ j)
      case False
      then have i + 1 = Suc j
        using Suc(2)
        by auto
      then show ?thesis
        using ‹?a i > 0› ‹?a (i + 1) > 0› *
        by auto
    next
      case True
      then show ?thesis
        using Suc(1) *
        by (smt Suc-eq-plus1 add-Suc-right one-add-one)
    qed
  qed
qed

then have ∀ j ≥ i+2. ?a j > 1
 by (metis Suc-eq-plus1 add-Suc-right le-iff-add one-add-one plus-nat.simps(2))

have *: ∀ j ≥ i+2. ?a (j + 2) > ?a (j + 1)
proof safe
  fix j
  assume i + 2 ≤ j
  then have ?a j > 1 ?a (j + 1) > 1
    using ‹∀ j ≥ i + 2. ?a j > 1› ‹i + 2 ≤ j›
    by auto
  then have ?a (j + 1) < ?a j * ?a (j + 1)
    by simp
  then show ?a (j + 2) > ?a (j + 1)
    using eq
    by smt
qed

have ∀ j > i + 3. ?a j > ?a (i + 3)
proof safe
```

fix j
assume $i + 3 < j$
then show $a\ ((i + 3)\ mod\ n) < a\ (j\ mod\ n)$
proof (*induction* j)
case *0*
then show *?case*
by *simp*
next
case (*Suc* j)
show *?case*
proof (*cases* $i + 3 < j$)
case *True*
then have *?a* $(i + 3) <$ *?a* j
using *Suc*
by *simp*
also have *?a* $j <$ *?a* $(j + 1)$
using *Suc*(*2*)
using $*$[*rule-format*, *of* $j-1$]
by *simp*
finally
show *?thesis*
by *simp*
next
case *False*
then have $i + 3 = j$
using *Suc*(*2*)
by *simp*
then show *?thesis*
using $*$[*rule-format*, *of* $i+2$]
by (*metis One-nat-def Suc-1 Suc-eq-plus1 add-Suc-right less-or-eq-imp-le numeral-3-eq-3*)
qed
qed
qed

then have *?a* $(i + 3 + n) >$ *?a* $(i + 3)$
by (*meson assms less-add-same-cancel1 less-le-trans zero-less-numeral*)

moreover

```
  have ?a (i + 3 + n) = ?a (i + 3)
    by simp

  ultimately

  show False
    by simp
qed

have no-zero: ∀ i. ?a i ≠ 0
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain i where ?a i = 0
    by auto
  then have ?a (i + n) = 0
    by auto
  have ?a (i + n + 2) = 1
    using ‹?a (i + n) = 0› eq
    by (metis add.commute mult-zero-left nat-arith.rule0)
  moreover
  have ?a (i + n + 1) = 1
    using ‹?a (i + n) = 0› eq[rule-format, of i+n−1] ‹n ≥ 3›
    by simp
  ultimately
  show False
    using no-pos-pos
    by (smt add.assoc one-add-one)
qed

have neg-neg-pos: ∀ i. ?a i < 0 ∧ ?a (i + 1) < 0 ⟶ ?a (i + 2) > 1
  using eq
  by (smt mult-neg-neg)

{
  fix i
  assume ?a i < 0 ?a (i + 1) < 0
  then have ?a (i + 2) > 1
    using neg-neg-pos
    by simp
  then have ?a (i + 3) < 0
```

```
    using no-pos-pos no-zero
    by (smt One-nat-def Suc-eq-plus1 add-Suc-right numeral-3-eq-3 one-add-one)

  have ?a (i + 4) < 1
  proof−
    have ?a (i + 4) = ?a (i+2) * ?a (i+3) + 1
      using eq[rule-format, of i+2]
      by (simp add: numeral-3-eq-3 numeral-Bit0)
    moreover
    have ?a (i+2) * ?a (i + 3) < 0
      using ‹?a (i + 3) < 0› ‹?a (i + 2) > 1›
      by (simp add: mult-pos-neg)
    ultimately
    show ?thesis
      by simp
  qed

  then have ?a (i + 4) < ?a (i + 2)
    using ‹?a (i + 2) > 1›
    by simp

  have ?a (i+5) − ?a (i+4) = (?a (i+3) * ?a (i+4) + 1) − (?a (i+3) * ?a
(i+2) + 1)
    using eq
    by (simp add: Groups.mult-ac(2) numeral-eq-Suc)
  also have ... = ?a (i+3) * (?a (i+4) − ?a (i+2))
    by (simp add: field-simps)
  finally have ?a (i+5) − ?a (i+4) > 0
    using ‹?a (i + 4) < ?a (i + 2)› ‹?a (i + 3) < 0›
    by (smt mult-neg-neg)
  then have ?a (i + 5) > ?a (i + 4)
    by auto
  then have ?a (i + 4) < 0
    using no-pos-pos no-zero
    by (smt Suc-eq-plus1 add-Suc-right numeral-eq-Suc pred-numeral-simps(3))

  have ?a (i+2) > 0 ∧ ?a (i+3) < 0 ∧ ?a (i+4) < 0
    using ‹1 < a ((i + 2) mod n)› ‹a ((i + 3) mod n) < 0› ‹a ((i + 4) mod n)
< 0›
    by simp
```

```
} note after-neg-neg = this

have ∃ i. ?a i < 0 ∧ ?a (i + 1) < 0
proof (rule ccontr)
  assume ¬ ?thesis
  then have alt: ∀ i. ?a i < 0 ⟷ ?a (i + 1) > 0
    using no-zero no-pos-pos
    by smt

  have neg: ∀ i k. ?a i < 0 ⟶ ?a (i + 2*k) < 0
  proof safe
    fix i k
    assume ?a i < 0
    then show ?a (i + 2 * k) < 0
    proof (induction k)
      case 0
      then show ?case
        by simp
    next
      case (Suc k)
      then show ?case
        using alt
        by (smt add.assoc add.commute mult-Suc-right no-zero one-add-one)
    qed
  qed

  have inc: ∀ i. ?a i < 0 ⟶ ?a i < ?a (i + 2)
  proof safe
    fix i
    assume ?a i < 0
    have ?a (i+1) > 0
      using alt
      using ‹?a i < 0›
      by blast
    then have ?a (i+2) < 0
      using alt
      by (smt add.assoc no-zero one-add-one)
    then have ?a (i+3) > 0
      using alt
      by (simp add: numeral-3-eq-3)
```

```
  have ?a i * ?a (i+1) + 1 < ?a (i+1) * ?a (i+2) + 1
    using ‹?a (i+2) < 0› ‹?a (i+3) > 0› eq
    by (simp add: numeral-eq-Suc)
  then show ?a i < ?a (i + 2)
    using ‹?a (i + 1) > 0›
  by (smt Groups.mult-ac(2) Suc-eq-plus1 add-2-eq-Suc' alt eq mult-less-cancel-left1)
qed

obtain i where ?a i < 0
  using alt
  by (meson linorder-neqE-linordered-idom no-zero)
have ∀ k ≥ 1. ?a i < ?a (i + 2*k)
proof safe
  fix k::nat
  assume 1 ≤ k
  then show ?a i < ?a (i + 2*k)
  proof (induction k)
    case 0
    then show ?case
      by simp
  next
    case (Suc k)
    show ?case
    proof (cases k = 0)
      case True
      then show ?thesis
        using inc ‹?a i < 0›
        by auto
    next
      case False
      then show ?thesis
        using ‹?a i < 0›
        using Suc(1) inc[rule-format, of i + 2*k] neg[rule-format, of i k]
        by simp
    qed
  qed
qed
then have ?a i < ?a (i + 2*n)
  using ‹n ≥ 3›
  by (simp add: numeral-eq-Suc)
```

```
    then show False
      by simp
  qed

  then obtain i where ?a i < 0 ?a (i + 1) < 0
    by auto

  have neg-neg-pos: ∀ k. ?a (i + 3 * k) < 0 ∧ ?a (i + 1 + 3*k) < 0 ∧ ?a (i +
2 + 3*k) > 0 (is ∀ k. ?P k)
  proof
    fix k
    show ?P k
    proof (induction k)
      case 0
      then show ?case
        using ‹?a i < 0› ‹?a (i + 1) < 0› after-neg-neg[of i]
        by simp
    next
      case (Suc k)
      then show ?case
        using after-neg-neg[of i + 3*k]
        using after-neg-neg[of i + 3*k + 3]
        by (simp add: numeral-3-eq-3 numeral-Bit0)
    qed
  qed

  show 3 dvd n
  proof−
    have n mod 3 = 0 ∨ n mod 3 = 1 ∨ n mod 3 = 2
      by auto
    then show ?thesis
    proof
      assume n mod 3 = 0
      then show ?thesis
        by auto
    next
      assume n mod 3 = 1 ∨ n mod 3 = 2
      then have False
      proof
        assume n mod 3 = 1
```

```
      then obtain k where n = 3 * k + 1
      by (metis add-diff-cancel-left' add-diff-cancel-right' add-eq-if assms dvd-minus-mod
dvd-mult-div-cancel not-numeral-le-zero plus-1-eq-Suc)
      then have ?a (i + 1) = ?a (i + 2 + 3*k)
      by (metis add.assoc add-Suc-right mod-add-self2 one-add-one plus-1-eq-Suc)
      then show False
        using neg-neg-pos[rule-format, of 0] neg-neg-pos[rule-format, of k]
        by simp
    next
      assume n mod 3 = 2
      then obtain k where n = 3 * k + 2
      by (metis One-nat-def Suc-1 add.commute add-Suc-shift add-diff-cancel-left'
assms dvd-minus-mod dvd-mult-div-cancel le-iff-add numeral-3-eq-3)
      then have ?a i = ?a (i + 2 + 3*k)
      by (metis add.assoc add-Suc-right mod-add-self2 one-add-one plus-1-eq-Suc)
      then show False
        using neg-neg-pos[rule-format, of 0] neg-neg-pos[rule-format, of k]
        by simp
    qed
    then show ?thesis
      by simp
  qed
 qed
qed

end
```

5.1.2 IMO 2018 SL - A4

```
theory IMO-2018-SL-A4-sol
imports Complex-Main
begin

definition is-Max :: 'a::linorder set ⇒ 'a ⇒ bool where
  is-Max A x ⟷ x ∈ A ∧ (∀ x' ∈ A. x' ≤ x)

lemma sum-list-cong:
  assumes ⋀ x. x ∈ set l ⟹ f x = g x
  shows (∑ x ← l. f x) = (∑ x ← l. g x)
  using assms
```

```
  by (metis map-eq-conv)

lemma Max-ge-Min:
  assumes finite A A ≠ {}
  shows Max A ≥ Min A
  using assms
  by simp

theorem IMO2018SL-A4:
  shows
  is-Max {a 2018 − a 2017 | a::nat ⇒ real. a 0 = 0 ∧ a 1 = 1 ∧ (∀ n ≥ 2. ∃
k. 1 ≤ k ∧ k ≤ n ∧ a n = (∑ i ← [n−k..<n]. a i) / real k)}
   (2016 / 2017^2) (is is-Max {?f a | a. ?P a} ?m)
  unfolding is-Max-def
proof
  show ?m ∈ {?f a | a. ?P a}
  proof−
    let ?a = (λ n. if n = 0 then 0
                   else if n < 2017 then 1
                   else if n = 2017 then 1 − 1/2017
                   else 1 − 1/2017^2) :: (nat ⇒ real)
    have ?P ?a
    proof safe
      show ?a 0 = 0
        by simp
    next
      show ?a 1 = 1
        by simp
    next
      fix n::nat
      assume 2 ≤ n

      show ∃ k. 1 ≤ k ∧ k ≤ n ∧ ?a n = (∑ i←[n − k..< n]. ?a i) / real k
      proof (cases n < 2017)
        case True
        have [n−1..<n] = [n−1]
          using ⟨n ≥ 2⟩
          by (simp add: upt-rec)
        then show ?thesis
          using ⟨n ≥ 2⟩ ⟨n < 2017⟩
```

```
      by (rule-tac x=1 in exI, auto)
    next
      case False
      show ?thesis
      proof (cases n = 2017)
        case True
        have [0..<2017] = [0] @ [1..<2017]
        by (metis One-nat-def less-numeral-extra(4) numeral-eq-Suc plus-1-eq-Suc
upt-add-eq-append upt-rec zero-le-one zero-less-one)
        then have (∑ i←[0..<2017]. ?a i) = ?a 0 + (∑ i←[1..<2017]. ?a i)
          by simp
       then have (∑ i←[0..<2017]. ?a i) = (∑ i←[0..<1]. 0) + (∑ i←[1..<2017].
1)
          using sum-list-cong[of [1..<2017] ?a λ k. 1]
          by auto
        then have (∑ i←[0..<2017]. ?a i) = 2016
          by (simp add: sum-list-triv)
        then show ?thesis
          using ‹n = 2017›
          by (rule-tac x=2017 in exI, auto)
      next
        case False
        show ?thesis
        proof (cases n = 2018)
          case True
          have [1..<2018] = [1..<2017] @ [2017]
        by (metis one-le-numeral one-plus-numeral plus-1-eq-Suc semiring-norm(4)
semiring-norm(5) upt-Suc-append)
            then have (∑ i←[1..<2018]. ?a i) = (∑ i←[1..<2017]. ?a i) + ?a
2017
            using sum-list-append[of [1..<2017] [2017..<2018]]
            by simp
          then have (∑ i←[1..<2018]. ?a i) = 2016 + (1 - 1/2017)
            using sum-list-cong[of [1..<2017] ?a λ k. 1]
            by (simp add: sum-list-triv)
          then show ?thesis
            using ‹n = 2018›
            by (rule-tac x=2017 in exI, auto)
        next
          case False
```

```
          have [n−1..<n] = [n−1]
            using ⟨n ≥ 2⟩
            by (simp add: upt-rec)
          then show ?thesis
            using ⟨¬ n < 2017⟩ ⟨n ≠ 2017⟩ ⟨n ≠ 2018⟩ ⟨n ≥ 2⟩
            by (rule-tac x=1 in exI, auto)
        qed
      qed
    qed
    moreover
    have ?f ?a = ?m
      by simp
    ultimately
    show ?thesis
      by (smt mem-Collect-eq)
  qed
next
  show ∀ x′∈{?f a | a. ?P a}. x′ ≤ ?m
  proof safe
    fix a :: nat ⇒ real
    let ?S = λ n k. (∑ i ← [n−k..<n]. a i)
    assume a 0 = 0 a 1 = 1 and ∗: ∀n≥2. ∃k≥1. k ≤ n ∧ a n = ?S n k / real
k
    let ?A = λ n. {?S n k / k | k. k ∈ {1..<n+1}}
    let ?max = λ n. Max (?A n)
    let ?min = λ n. Min (?A n)
    let ?Δ = λ n. ?max n − ?min n

    have A: ∀ n ≥ 1. finite (?A n) ∧ ?A n ≠ {}
      by auto

    have ∀ n ≥ 2. ?Δ n ≥ 0
    proof safe
      fix n::nat
      assume 2 ≤ n
      then have ?max n ≥ ?min n
        using Max-ge-Min[of ?A n] A[rule-format, of n]
        by force
      then show ?Δ n ≥ 0
```

```
    by simp
qed

have ∀ n ≥ 2. ?min n ≤ a n ∧ a n ≤ ?max n
proof safe
  fix n::nat
  assume n ≥ 2
  then have n ≥ 1
    by simp
  have a n ∈ ?A n
    using ∗ ‹n ≥ 2›
    by force
  then show ?min n ≤ a n a n ≤ ?max n
    using A[rule-format, OF ‹n ≥ 1›]
    using Min-le[of ?A n a n] Max-ge[of ?A n a n]
    by blast+
qed

have ∀ n ≥ 2. a (n − 1) ∈ ?A n
proof safe
  fix n::nat
  assume n ≥ 2
  then have [n−1..<n] = [n−1]
    using upt-rec by auto
  then have a (n − 1) = ?S n 1
    by simp
  then show ∃ k. a (n − 1) = ?S n k / k ∧ k ∈ {1..<n+1}
    using ‹n ≥ 2›
    by force
qed

have ∀ n ≥ 2. ?min n ≤ a (n−1) ∧ a (n−1) ≤ ?max n
proof safe
  fix n::nat
  assume n ≥ 2
  then have n ≥ 1
    by simp
  have a (n − 1) ∈ ?A n
    using ‹∀ n ≥ 2. a (n − 1) ∈ ?A n› ‹n ≥ 2›
    by force
```

```
    then show ?min n ≤ a (n − 1) a (n − 1) ≤ ?max n
      using A[rule-format, OF ⟨n ≥ 1⟩]
      using Min-le[of ?A n a (n − 1)] Max-ge[of ?A n a (n − 1)]
      by blast+
  qed

  have ?f a ≤ ?Δ 2018
    using ⟨∀ n ≥ 2. ?min n ≤ a n ∧ a n ≤ ?max n⟩[rule-format, of 2018]
    using ⟨∀ n ≥ 2. ?min n ≤ a (n−1) ∧ a (n−1) ≤ ?max n⟩[rule-format, of
2018]
    by auto

  have Claim1: ∀ n > 2. ?Δ n ≤ (n−1)/n * ?Δ (n−1)
  proof safe
    fix n::nat
    assume 2 < n
    then have 1 ≤ n
      by simp
    obtain k where ?max n = ?S n k / k 1 ≤ k k ≤ n
      using A[rule-format, OF ⟨1 ≤ n⟩] Max-in[of ?A n]
      by force
    obtain l where ?min n = ?S n l / l 1 ≤ l l ≤ n
      using A[rule-format, OF ⟨1 ≤ n⟩] Min-in[of ?A n]
      by force

    have [n − k..<n] = [n − 1 − (k − 1)..<n − 1] @ [n − 1]
      using ⟨1 ≤ k⟩ ⟨k ≤ n⟩ ⟨1 ≤ n⟩
    by (metis Nat.diff-diff-eq diff-le-self le-add-diff-inverse plus-1-eq-Suc upt-Suc-append)
    then have ?S n k = ?S (n−1) (k−1) + a (n−1)
      by simp

    have [n − l..<n] = [n − 1 − (l − 1)..<n − 1] @ [n − 1]
      using ⟨1 ≤ l⟩ ⟨l ≤ n⟩ ⟨1 ≤ n⟩
    by (metis Nat.diff-diff-eq diff-le-self le-add-diff-inverse plus-1-eq-Suc upt-Suc-append)
    then have ?S n l = ?S (n−1) (l−1) + a (n−1)
      by simp

    have real (k − Suc 0) = real k − 1
      using ⟨k ≥ 1⟩
      by simp
```

```
have ?S (n−1) (k−1) ≤ (k − 1) ∗ ?max (n − 1)
proof (cases k = 1)
  case True
  then show ?thesis
    by simp
next
  case False
  have n−1 ≥ 1
    using ‹n > 2›
    by simp
  have ?S (n−1) (k−1) / (k − 1) ≤ ?max (n − 1)
  proof (rule Max-ge)
    show finite (?A (n−1))
      using A[rule-format, OF ‹n−1 ≥ 1›]
      by simp
  next
    show ?S (n−1) (k−1) / (k − 1) ∈ ?A (n−1)
      using ‹k ≠ 1› ‹k ≥ 1› ‹k ≤ n›
      by simp (rule-tac x=k−1 in exI, auto)
  qed
  then show ?thesis
    using ‹k ≥ 1› ‹k ≠ 1›
    by (simp add: field-simps)
qed

have ?S (n−1) (l−1) ≥ (l − 1) ∗ ?min (n − 1)
proof (cases l = 1)
  case True
  then show ?thesis
    by simp
next
  case False
  have n−1 ≥ 1
    using ‹n > 2›
    by simp
  have ?S (n−1) (l−1) / (l − 1) ≥ ?min (n − 1)
  proof (rule Min-le)
    show finite (?A (n−1))
      using A[rule-format, OF ‹n−1 ≥ 1›]
```

```
        by simp
      next
        show ?S (n−1) (l−1) / (l − 1) ∈ ?A (n−1)
          using ‹l ≠ 1› ‹l ≥ 1› ‹l ≤ n›
          by simp (rule-tac x=l−1 in exI, auto)
      qed
      then show ?thesis
        using ‹l ≥ 1› ‹l ≠ 1›
        by (simp add: field-simps)
    qed

    have ?min (n−1) ≤ a (n−1) a (n−1) ≤ ?max (n−1)
      using ‹∀ n ≥ 2. ?min n ≤ a n ∧ a n ≤ ?max n›[rule-format, of n−1] ‹n
> 2›
      by simp-all

    {
      fix x1 x2::real
      assume 0 < x1 x1 ≤ x2
      then have (x1 − 1) / x1 ≤ (x2 − 1) / x2
        by (metis (no-types, hide-lams) diff-divide-distrib diff-mono divide-self-if
frac-le leD order-refl zero-le-one)
    } note mono = this


    have k∗(?max n − a (n−1)) = ?S n k − k ∗ a (n−1)
      using ‹?max n = ?S n k / k›
      by (simp add: algebra-simps)
    also have ... = ?S (n−1) (k−1) − (real k − 1) ∗ a (n−1)
      using ‹?S n k = ?S (n−1) (k−1) + a (n−1)›
      by (simp add: field-simps)
    also have ... ≤ (k − 1) ∗ ?max (n − 1) − (real k − 1) ∗ a (n−1)
      using ‹?S (n−1) (k−1) ≤ (k − 1) ∗ ?max (n − 1)›
      by simp
    also have ... = (real k − 1) ∗ (?max (n − 1) − a (n−1))
      using ‹k ≥ 1›
      by (auto simp add: right-diff-distrib)
    finally have k∗(?max n − a (n−1)) ≤ (real k − 1) ∗ (?max (n − 1) − a
(n−1))
      .
```

```
      then have ?max n − a (n−1) ≤ (real k − 1) / k ∗ (?max (n−1) − a
(n−1))
       using ⟨k ≥ 1⟩
       by (simp add: field-simps)
      also have (real k − 1) / k ∗ (?max (n−1) − a (n−1)) ≤
                (real n − 1) / n ∗ (?max (n−1) − a (n−1))
      proof−
       have (real k − 1) / k ≤ (real n − 1) / n
        using mono[of real k real n] ⟨k ≤ n⟩ ⟨k ≥ 1⟩
        by simp
       then show ?thesis
        using ⟨a (n − 1) ≤ ?max (n−1)⟩
        by (smt mult-cancel-right real-mult-le-cancel-iff1)
      qed
      finally
      have 1: ?max n − a (n−1) ≤ (real n − 1) / n ∗ (?max (n−1) − a (n−1))
       .

      have l ∗ (a (n−1) − ?min n) = l ∗ a (n−1) − ?S n l
       using ⟨?min n = ?S n l / l⟩
       by (simp add: algebra-simps)
      also have ... = (real l − 1) ∗ a (n−1) − ?S (n−1) (l−1)
       using ⟨?S n l = ?S (n−1) (l−1) + a (n−1)⟩
       by (simp add: field-simps)
      also have ... ≤ (real l − 1) ∗ a (n−1) − (l − 1) ∗ ?min (n − 1)
       using ⟨?S (n−1) (l−1) ≥ (l − 1) ∗ ?min (n − 1)⟩
       by (simp add: field-simps)
      also have ... = (real l − 1) ∗ (a (n−1) − ?min (n − 1))
       using ⟨l ≥ 1⟩
       by (auto simp add: right-diff-distrib)
      finally have l∗(a (n−1) − ?min n) ≤ (real l − 1) ∗ (a (n−1) − ?min (n
− 1))
       .
      then have a (n−1) − ?min n ≤ (real l − 1) / l ∗ (a (n−1) − ?min (n−1))
       using ⟨l ≥ 1⟩
       by (simp add: field-simps)
      also have (real l − 1) / l ∗ (a (n−1) − ?min (n−1)) ≤
                (real n − 1) / n ∗ (a (n−1) − ?min (n−1))
      proof−
       have (real l − 1) / l ≤ (real n − 1) / n
```

```
      using mono[of real l real n] ‹l ≤ n› ‹l ≥ 1›
      by simp
    then show ?thesis
      using ‹a (n − 1) ≥ ?min (n−1)›
      by (smt mult-cancel-right real-mult-le-cancel-iff1)
  qed
  finally
  have 2: a (n−1) − ?min n ≤ (real n − 1) / n ∗ (a (n−1) − ?min (n−1))
    .

  have ?Δ n = (?max n − a (n−1)) + (a (n−1) − ?min n)
    by simp
  also have ... ≤ (real n − 1) / n ∗ ((?max (n−1) − a (n−1)) + (a (n−1)
− ?min (n−1)))
    using 1 2
    by (simp add: right-diff-distrib′)
  finally show ?Δ n ≤ (real n − 1) / n ∗ ?Δ (n−1)
    by simp
qed

obtain Δ where Δ = ?Δ by auto
then have Claim1′: ∀ n > 2. Δ n ≤ (n−1)/n ∗ Δ (n−1)
  using Claim1
  by blast

have Claim1-iter′: ⋀ N q. ⟦2 ≤ q; q ≤ N⟧ ⟹ Δ (N+1) ≤ Δ (q+1) ∗ (q +
1) / (N + 1)
proof−
  fix N q :: nat
  assume 2 ≤ q q ≤ N
  then show Δ (N+1) ≤ Δ (q+1) ∗ (q + 1) / (N + 1)
  proof (induction N)
    case 0
    then show ?case
      by simp
  next
    case (Suc N)
    show ?case
    proof (cases q ≤ N)
      case True
```

```
        have Δ (N + 2) ≤ ((N + 1)/(N + 2)) * Δ (N + 1)
          using Claim1'[rule-format, of Suc N + 1] ‹2 ≤ q› ‹q ≤ N›
          by simp
        moreover
        have Δ (N + 1) ≤ Δ (q + 1) * (q + 1) / (N + 1)
          using True ‹2 ≤ q› Suc(1)
          by simp
        then have ((N + 1)/(N + 2)) * Δ (N + 1) ≤ ((N + 1)/(N + 2)) *
(Δ (q + 1) * (q + 1) / (N + 1))
          by (subst real-mult-le-cancel-iff2, simp-all)
        ultimately
        show ?thesis
          by simp
      next
        case False
        then have q = N+1
          using Suc(3)
          by simp
        then show ?thesis
          by simp
      qed
    qed
  qed

  {
    fix q::nat
    assume ∀ n. 1 ≤ n ∧ n < q ⟶ a n = 1

    have ∀ k. 1 ≤ k ∧ k < q ⟶ ?S q k = k
    proof safe
      fix k::nat
      assume 1 ≤ k k < q
      then have (∑ i ← [q−k..<q]. a i) = (∑ i ← [q−k..<q]. 1)
        using sum-list-cong[of [q−k..<q] a λ i. 1]
        using ‹∀ n. 1 ≤ n ∧ n < q ⟶ a n = 1› ‹k < q›
        by fastforce
      then show ?S q k = k
        using ‹1 ≤ k› ‹k < q›
        by (simp add: sum-list-triv)
    qed
```

```
  }
  note all-1-Sqk = this

  {
    fix q::nat
    assume q ≥ 2
    assume ∀ n. 1 ≤ n ∧ n < q ⟶ a n = 1
    have ?S q q = q − 1
    proof−
      have [q−q..<q] = [0] @ [1..<q]
        using ⟨2 ≤ q⟩
        using upt-rec by auto
      then have ?S q q = (∑ i ← [1..<q]. a i)
        using ⟨a 0 = 0⟩
        by auto
      also have ... = (∑ i ← [1..<q]. 1::real)
        using sum-list-cong[of [1..<q] a λ i. 1]
        using ⟨∀ n. 1 ≤ n ∧ n < q ⟶ a n = 1⟩
        by simp
      finally show ?thesis
        by (simp add: sum-list-triv)
    qed
  } note all-1-Sqq = this

  show ?f a ≤ ?m
  proof (cases ∀ n. 2 ≤ n ∧ n ≤ 2017 ⟶ a n = 1)
    case True
    then have ∀n. 1 ≤ n ∧ n < 2018 ⟶ a n = 1
      using ⟨a 1 = 1⟩
    by (metis Suc-leI add-le-cancel-left le-eq-less-or-eq one-add-one one-plus-numeral
plus-1-eq-Suc semiring-norm(4) semiring-norm(5))
    then have ∀ k. 1 ≤ k ∧ k ≤ 2018 ⟶ ?S 2018 k ≤ k
      using all-1-Sqk[of 2018] all-1-Sqq[of 2018]
    by (smt Suc-leI le-eq-less-or-eq of-nat-1 of-nat-diff one-add-one one-less-numeral-iff
plus-1-eq-Suc semiring-norm(76))
    then have a 2018 ≤ 1
      using ∗[rule-format, of 2018]
      by auto
    then show ?thesis
      using True
```

```
      by auto
    next
      case False
      let ?Q = {q. 2 ≤ q ∧ q ≤ 2017 ∧ a q ≠ 1}
      let ?q = Min ?Q
      have ?Q ≠ {}
        using False ⟨a 1 = 1⟩
        by auto
      then have 2 ≤ ?q ?q ≤ 2017 a ?q ≠ 1
        using Min-in[of ?Q]
        by auto

      have ∀ n. 2 ≤ n ∧ n < ?q ⟶ a n = 1
      proof (rule ccontr)
        assume ¬ ?thesis
        then obtain n where 2 ≤ n n < ?q a n ≠ 1
          by auto
        then have n ∈ ?Q
          using ⟨?q ≤ 2017⟩
          by auto
        then show False
          using Min-le[of ?Q n] ⟨?Q ≠ {}⟩ ⟨a n ≠ 1⟩ ⟨n < ?q⟩
          by auto
      qed

      obtain q where q = ?q 2 ≤ q q ≤ 2017 using ⟨2 ≤ ?q⟩ ⟨?q ≤ 2017⟩ by
auto
      then have ∀ n. 1 ≤ n ∧ n < q ⟶ a n = 1
        using ⟨∀ n. 2 ≤ n ∧ n < ?q ⟶ a n = 1⟩ ⟨a 1 = 1⟩
        by (metis Suc-1 Suc-leI le-eq-less-or-eq)
      then have ∀ k. 1 ≤ k ∧ k < q ⟶ ?S q k = k  ?S q q = q − 1
        using all-1-Sqk[of q] all-1-Sqq[of q] ⟨2 ≤ q⟩
        by simp-all
      then have ∀ k. 1 ≤ k ∧ k ≤ q ⟶ ?S q k ≤ k
        using le-eq-less-or-eq
        by auto
      then have a q ≤ 1
        using ∗[rule-format, OF ⟨2 ≤ q⟩]
        by auto
      then have a q < 1
```

```
    using ⟨q = ?q⟩ ⟨a ?q ≠ 1⟩
    by auto

  have a q = ?S q q / q
    using ∗[rule-format, OF ⟨2 ≤ q⟩] ⟨a q < 1⟩ ⟨∀ k. 1 ≤ k ∧ k < q ⟶ ?S q
k = k⟩
  by (metis div-by-1 less-le of-nat-1 of-nat-le-iff one-eq-divide-iff order-class.order.antisym
zero-le-one)

  then have a q = 1 − 1/q
    using ⟨?S q q = q − 1⟩
    using ⟨q ≥ 2⟩
    by (simp add: field-simps)

  have ∀ i. 1 ≤ i ∧ i ≤ q ⟶ ?S (q+1) i = i − 1/q
  proof safe
    fix i
    assume 1 ≤ i i ≤ q
    show ?S (q+1) i = i − 1/q
    proof (cases i = 1)
      case True
      then show ?thesis
        using ⟨a q = 1 − 1/q⟩
        by simp
    next
      case False
      then have ?S (q+1) i = a q + ?S q (i−1)
        using ⟨1 ≤ i⟩ ⟨i ≤ q⟩
        by auto
      moreover
      have ?S q (i−1) = (i−1)
        using ⟨∀ k. 1 ≤ k ∧ k < q ⟶ ?S q k = k⟩[rule-format, of i−1]
        using ⟨1 ≤ i⟩ ⟨i ≤ q⟩ ⟨i ≠ 1⟩
        using Suc-le-eq
        by auto
      ultimately
      show ?thesis
        using ⟨a q = 1 − 1/q⟩ ⟨1 ≤ i⟩
        by simp
    qed
```

```
    qed

    have ?S (q+1) (q+1) = q - 1/q
    proof-
      have ?S (q+1) (q+1) = a q + ?S q q
        by simp
      then show ?thesis
        using ⟨?S q q = q - 1⟩ ⟨a q = 1 - 1/q⟩
        using ⟨2 ≤ q⟩
        by simp
    qed

    have qq: (real q - 1 / real q) / (real q + 1) = (real q - 1) / real q
    proof-
      have (real q + 1) * ((real q - 1 / real q) / (real q + 1)) = (real q + 1) *
((real q - 1) / real q)
        using ⟨2 ≤ q⟩
        by simp (simp add: field-simps)
      then show ?thesis
        by (subst (asm) mult-left-cancel, simp-all)
    qed

    have ?min (q+1) = (real q - 1)/real q  (is ?lhs = ?mn)
    proof (subst Min-eq-iff)
      show finite (?A (q+1))
        by simp
    next
      show ?A (q+1) ≠ {}
        using ⟨q ≥ 2⟩
        by auto
    next
      show ?mn ∈ ?A (q+1) ∧ (∀ m' ∈ ?A (q+1). m' ≥ ?mn)
      proof
        have ?mn = 1 - 1/q
          using ⟨2 ≤ q⟩
          by (simp add: field-simps)
        then have ?mn = ?S (q+1) 1
          using ⟨∀ i. 1 ≤ i ∧ i ≤ q ⟶ ?S (q+1) i = i - 1/q⟩[rule-format, of
1] ⟨2 ≤ q⟩
          by simp
```

```
    then show ?mn ∈ ?A (q+1)
      by force
    show ∀ m' ∈ ?A (q+1). m' ≥ ?mn
    proof
      fix m'
      assume m' ∈ ?A (q+1)
      then obtain k where k ∈ {1..<q+1+1} m' = ?S (q+1) k / k
        by force
      show m' ≥ ?mn
      proof (cases k ≤ q)
        case True
        then have m' = (k - 1/q) / k
          using ‹k ∈ {1..<q+1+1}› ‹m' = ?S (q+1) k / k›
          using ‹∀ i. 1 ≤ i ∧ i ≤ q ⟶ ?S (q+1) i = i - 1/q›
          by auto
        then have m' = 1 - 1/(q*k)
          using ‹k ∈ {1..<q+1+1}› ‹q ≥ 2›
          by (simp add: field-simps)
        then show ?thesis
          using ‹?mn = 1 - 1/q› ‹k ∈ {1..<q+1+1}› ‹2 ≤ q›
          by simp (simp add: field-simps)
      next
        case False
        then have k = q+1
          using ‹k ∈ {1..<q+1+1}›
          by simp
        then have m' = (real q - 1) / real q
          using ‹m' = ?S (q+1) k / k› ‹?S (q+1) (q+1) = q - 1/q›
          using qq
          by (metis of-nat-1 of-nat-add)
        then show ?thesis
          by simp
      qed
    qed
  qed
qed

moreover

have ?max (q+1) = ((real q) ^ 2 - 1)/(real q) ^ 2 (is ?lhs = ?mx)
```

```
proof (subst Max-eq-iff)
  show finite (?A (q+1))
    by simp
next
  show ?A (q+1) ≠ {}
    using ‹q ≥ 2›
    by auto
next
  show ?mx ∈ ?A (q+1) ∧ (∀ m'∈ ?A (q+1). m' ≤ ?mx)
  proof
    have ?mx = (?S (q+1) q) / q
      using ‹∀ i. 1 ≤ i ∧ i ≤ q ⟶ ?S (q+1) i = i - 1/q›[rule-format, of
q] ‹2 ≤ q›
      by simp (simp add: field-simps power2-eq-square)
    moreover
    have q ∈ {1..<q + 1 + 1}
      using ‹q ≥ 2›
      by simp
    ultimately
    show ?mx ∈ ?A (q+1)
      by force

    show ∀ m'∈ ?A (q+1). m' ≤ ?mx
    proof
      fix m'
      assume m' ∈ ?A (q+1)
      then obtain k where k ∈ {1..<q+1+1} m' = ?S (q+1) k / k
        by force
      show m' ≤ ?mx
      proof (cases k ≤ q)
        case True
        then have m' = (k - 1/q) / k
          using ‹k ∈ {1..<q+1+1}› ‹m' = ?S (q+1) k / k›
          using ‹∀ i. 1 ≤ i ∧ i ≤ q ⟶ ?S (q+1) i = i - 1/q›
          by auto
        then have m' = 1 - 1/(q*k)
          using ‹k ∈ {1..<q+1+1}› ‹q ≥ 2›
          by (simp add: field-simps)
        moreover
        have ?mx = 1 - 1/(q*q)
```

```
        using ⟨q ≥ 2⟩
        by (simp add: field-simps power2-eq-square)
      ultimately
      show ?thesis
        using ⟨k ≤ q⟩ ⟨2 ≤ q⟩ ⟨k ∈ {1..<q+1+1}⟩
        by simp (simp add: field-simps)
    next
      case False
      then have k = q+1
        using ⟨k ∈ {1..<q+1+1}⟩
        by simp
      then have m' = (real q − 1) / real q
        using ⟨m' = ?S (q+1) k / k⟩ ⟨?S (q+1) (q+1) = q − 1/q⟩ qq
        by (metis of-nat-1 of-nat-add)
      moreover
      have q ≤ q^2
        by (simp add: ⟨2 ≤ q⟩ power2-nat-le-imp-le)
      ultimately
      show ?thesis
        using ⟨2 ≤ q⟩
        by simp (simp add: field-simps)
    qed
  qed
 qed
qed

ultimately

have ?Δ (q+1) = ((real q)^2 − 1)/(real q)^2 − (real q − 1)/real q
 by simp
also have ... = (real q − 1)/(real q)^2
 using ⟨q ≥ 2⟩
 by (simp add: power2-eq-square field-simps)
finally have del: Δ (q+1) = (real q − 1)/(real q)^2
 using ⟨Δ = ?Δ⟩
 by simp
then have Δ (2017 + 1) ≤ (real q − 1) / (real q)^2 * real (q + 1) / 2018
 using Claim1-iter'[OF ⟨2 ≤ q⟩ ⟨q ≤ 2017⟩]
 by simp
also have ... = ((real q^2 − 1) / (real q)^2) / 2018
```

```
      by (simp add: field-simps power2-eq-square)
    also have ... = (1 - (1 / (real q)^2)) / 2018
      using ‹q ≥ 2›
      by (simp add: field-simps)
    also have ... ≤ (1 - (1 / 2017^2)) / 2018
    proof-
      have q^2 ≤ 2017^2
        using ‹2 ≤ q› ‹q ≤ 2017›
        using power-mono by blast
      then have (real q) ^2 ≤ 2017^2
        by (metis of-nat-le-iff of-nat-numeral of-nat-power)
      then show ?thesis
        using ‹2 ≤ q›
        by (simp add: field-simps power2-eq-square)
    qed
    finally have Δ 2018 ≤ ?m
      by simp

    then show ?thesis
      using ‹?f a ≤ ?Δ 2018› ‹Δ = ?Δ›
      by simp
  qed
 qed
qed

end
```

5.2 Combinatorics problems

5.2.1 IMO 2018 SL - C1

```
theory IMO-2018-SL-C1-sol
imports Complex-Main
begin

lemma sum-geom-nat:
  fixes q::nat
  assumes q > 1
  shows (∑k∈{0..<n}. q^k) = (q^n - 1) div (q - 1)
proof (induction n)
```

```
  case 0
  then show ?case by simp
next
  case (Suc n)
  then show ?case
    by (smt Nat.add-diff-assoc2 One-nat-def Suc-1 Suc-leI add.commute assms
div-mult-self4 le-trans mult-eq-if nat-one-le-power one-le-numeral power.simps(2)
sum.op-ivl-Suc zero-less-diff zero-order(3))
qed

declare [[smt-timeout = 20]]

lemma div-diff-nat:
  fixes a b c :: nat
  assumes c dvd a c dvd b
  shows (a − b) div c = a div c  − b div c
  using assms
  by (smt add-diff-cancel-left' div-add dvd-diff-nat le-iff-add nat-less-le neq0-conv
not-less zero-less-diff)

lemma sum-geom-nat':
  fixes q::nat
  assumes q > 1 m ≤ n
  shows (∑ k∈{m..<n}. q^k) = (q^n − q^m) div (q − 1)
  using assms
proof (induction n)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  show ?case
  proof (cases m ≤ n)
    case True
    then have sum ((^) q) {m..<Suc n} = (q ^ n − q ^ m) div (q − 1) + q^n
      using Suc
      by simp
    also have ... = ((q ^ n − q ^ m) + (q − 1) ∗ q^n) div (q − 1)
      using ‹q > 1›
      by auto
```

```
    also have ... = ((q ^ n - q ^ m) + (q^(n+1) - q^n)) div (q - 1)
      by (simp add: algebra-simps)
    also have ... = (q ^ (n+1) - q ^ m) div (q - 1)
      using True assms(1) by auto
    finally show ?thesis
      by simp
  next
    case False
    then have m = n + 1
      using Suc(3)
      by auto
    then show ?thesis
      by simp
  qed
qed


theorem IMO2018SL-C1:
  fixes n :: nat
  assumes n ≥ 3
  shows ∃ (S::nat set). card S = 2*n ∧ (∀ x ∈ S. x > 0) ∧
        (∀ m. 2 ≤ m ∧ m ≤ n ⟶ (∃ S1 S2. S1 ∩ S2 = {} ∧ S1 ∪ S2 = S ∧
card S1 = m ∧ ∑ S1 = ∑ S2))
proof−
  let ?Sa = {(3::nat)^k | k. k ∈ {1..<n}} and ?Sb = {2 * (3::nat)^k | k. k ∈
{1..<n}} and ?Sc = {1::nat, (3^n + 9) div 2 - 1}
  let ?S = ?Sa ∪ ?Sb ∪ ?Sc

  have finite ?Sa finite ?Sb finite ?Sc finite (?Sa ∪ ?Sb)
    by auto

  have ?Sa ∩ ?Sb = {}
  proof safe
    fix ka kb
    assume ka ∈ {1..<n} kb ∈ {1..<n} (3::nat)^ka = 2*3^kb
    have odd ((3::nat)^ka) even (2*3^kb)
      by simp-all
    then have False
      using ‹(3::nat)^ka = 2*3^kb›
      by simp
```

```
    then show 3^ka ∈ {}
      by simp
  qed

  have 1 < ((3::nat) ^ n + 9) div 2
    by linarith

  have ¬ 3 dvd (((3::nat) ^ n + 9) div 2 - 1)
  proof−
    have 3 dvd ((3::nat) ^ n + 9) div 2
    proof−
      have (3::nat) ^ n + 9 = (3^2) * (3::nat)^(n−2) + 9
        using ⟨n ≥ 3⟩
      by (metis One-nat-def add-leD2 le-add-diff-inverse numeral-3-eq-3 one-add-one
plus-1-eq-Suc power-add)
      then have (3::nat) ^ n + 9 = 9*(3^(n−2) + 1)
        by simp
      then have ((3::nat) ^ n + 9) div 2 = (9 * (3^(n−2) + 1)) div 2
        by auto
      then have ((3::nat) ^ n + 9) div 2 = 9 * ((3^(n−2) + 1) div 2)
      by (metis One-nat-def div-mult-swap dvd-mult-div-cancel even-add even-power
even-succ-div-two num.distinct(1) numeral-3-eq-3 numeral-eq-one-iff one-add-one
plus-1-eq-Suc)
      then show ?thesis
        by simp
    qed
    then show ?thesis
      using ⟨((3::nat) ^ n + 9) div 2 > 1⟩
    by (meson dvd-diffD1 less-imp-le-nat nat-dvd-1-iff-1 numeral-eq-one-iff semiring-norm(86))
  qed

  have (?Sa ∪ ?Sb) ∩ ?Sc = {}
  proof−
    have ?Sa ∩ ?Sc = {}
    proof safe
      fix k
      assume k ∈ {1..<n} (3::nat) ^ k = 1
      then show 3 ^ k ∈ {}
        by simp
    next
```

```
  fix k
  assume k ∈ {1..<n} (3::nat) ^ k = (3 ^ n + 9) div 2 - 1
  moreover
  have 3 dvd (3::nat) ^ k
    using ⟨k ∈ {1..<n}⟩
    by auto
  ultimately
  have False
    using ⟨¬ 3 dvd (3 ^ n + 9) div 2 - 1⟩
    by simp
  then show 3 ^ k ∈ {}
    by simp
qed

moreover

have ?Sb ∩ ?Sc = {}
proof safe
  fix k
  assume k ∈ {1..<n} 2 * (3::nat) ^ k = 1
  then show 2 * 3 ^ k ∈ {}
    by simp
next
  fix k
  assume k ∈ {1..<n} 2 * (3::nat) ^ k = (3 ^ n + 9) div 2 - 1
  moreover
  have 3 dvd 2 * (3::nat) ^ k
    using ⟨k ∈ {1..<n}⟩
    by auto
  ultimately
  have False
    using ⟨¬ 3 dvd (3 ^ n + 9) div 2 - 1⟩
    by simp
  then show 2 * 3 ^ k ∈ {}
    by simp
qed

ultimately
show ?thesis
  by blast
```

```
  qed

  show ?thesis
  proof (rule-tac x=?S in exI, safe)
    show card ?S = 2*n
    proof−
      have card (?Sa ∪ ?Sb) = (n − 1) + (n − 1)
      proof−
        have inj-on ((^) (3::nat)) {1..<n}
          unfolding inj-on-def
          by auto
        then have card ?Sa = n−1
          using card-image[of λ k. 3^k {1..<n}]
          by (smt Collect-cong Setcompr-eq-image card-atLeastLessThan)

        moreover

        have inj-on (λ k. 2 * (3::nat) ^ k) {1..<n}
          unfolding inj-on-def
          by auto

        then have card ?Sb = n−1
          using card-image[of λ k. 2 * 3^k {1..<n}]
          by (smt Collect-cong Setcompr-eq-image card-atLeastLessThan)


        ultimately
        show ?thesis
          using ⟨n ≥ 3⟩ card-Un-disjoint[of ?Sa ?Sb] ⟨?Sa ∩ ?Sb = {}⟩ ⟨finite ?Sa⟩
⟨finite ?Sb⟩
          by smt
      qed

      moreover

      have card {1, ((3::nat)^n + 9) div 2 − 1} = 2
        using ⟨1 < ((3::nat) ^ n + 9) div 2⟩
        by auto

      ultimately
```

```
    show card ?S = 2*n
      using ⟨n ≥ 3⟩ card-Un-disjoint[of ?Sa ∪ ?Sb ?Sc] ⟨(?Sa ∪ ?Sb) ∩ ?Sc =
{}⟩ ⟨finite (?Sa ∪ ?Sb)⟩ ⟨finite ?Sc⟩
      by (smt Nat.add-diff-assoc2 Suc-1 Suc-eq-plus1 add-Suc-right card-infinite
diff-add-inverse2 le-trans mult-2 nat.simps(3) one-le-numeral)
  qed
 next
  fix k
  assume k ∈ {1..<n}
  then show 0 < (3::nat) ^ k 0 < 2 * (3::nat) ^ k
    by simp-all
 next
  show 0 < ((3::nat) ^ n + 9) div 2 - 1
    using ⟨1 < (3 ^ n + 9) div 2⟩ zero-less-diff
    by blast
 next
  fix m
  assume 2 ≤ m m ≤ n
  let ?Am' = {2 * (3::nat)^k | k. k ∈ {n-m+1..<n}} and ?Am'' = {(3::nat)
^ (n-m+1)}
  let ?Am = ?Am' ∪ ?Am''
  let ?Bm = ?S - ?Am

  have ?Am' ⊆ ?Sb
    using ⟨m ≤ n⟩
    by auto

  have ?Am'' ⊆ ?Sa
    using ⟨m ≤ n⟩ ⟨2 ≤ m⟩
    by force


  have ?Am ∩ ?Bm = {}
    by blast

  moreover

  have Am: ?Am' ∩ ?Am'' = {}  finite ?Am' finite ?Am''
    using ⟨?Am' ⊆ ?Sb⟩ ⟨?Am'' ⊆ ?Sa⟩ ⟨?Sa ∩ ?Sb = {}⟩
```

```
    by auto

  have finite ?Am finite ?Bm
    by auto

  have ?Am ∪ ?Bm = ?S
  proof−
    have ?Am ⊆ ?S
      using ‹?Am' ⊆ ?Sb› ‹?Am'' ⊆ ?Sa›
      by blast
    then show ?thesis
      by blast
  qed

  moreover

  have card ?Am = m
  proof−
    have inj-on (λ k. 2 ∗ (3::nat) ^ k) {n−m+1..<n}
      unfolding inj-on-def
      by auto
    then show ?thesis
      using card-image[of λ k. 2 ∗ (3::nat) ^k {n−m+1..<n}]
          card-Un-disjoint[of ?Am' ?Am''] Am
      unfolding Setcompr-eq-image
     by (smt Int-insert-right-if1 One-nat-def Suc-eq-plus1 Un-insert-right ‹({2 ∗ 3
^ k |k. k ∈ {n − m + 1..<n}} ∪ {3 ^ (n − m + 1)}) ∩ ({3 ^ k |k. k ∈ {1..<n}}
∪ {2 ∗ 3 ^ k |k. k ∈ {1..<n}} ∪ {1, (3 ^ n + 9) div 2 − 1} − ({2 ∗ 3 ^ k |k. k ∈
{n − m + 1..<n}} ∪ {3 ^ (n − m + 1)})) = {}› ‹2 ≤ m› ‹m ≤ n› add.commute
add-diff-inverse-nat add-le-cancel-left card.insert card-atLeastLessThan card-empty
diff-Suc-Suc diff-diff-cancel disjoint-insert(2) finite.emptyI insertCI insert-absorb
le-trans linorder-not-le one-le-numeral)
  qed

  moreover
  have ∑ ?Am = ∑ ?Bm
  proof−
    have (∑ ?Am) = 3^n
    proof−
      have ∑ ?Am' = (∑k∈{n−m+1..<n}. 2∗3^k)
```

```
    proof−
      have inj-on (λ k. 2∗(3::nat)^k) {n−m+1..<n}
        unfolding inj-on-def
        by auto
      then show ?thesis
        unfolding Setcompr-eq-image
        by (simp add: sum.reindex-cong)
    qed
    also have ... = 2 ∗ (∑k∈{n−m+1..<n}. 3^k)
      by (simp add: sum-distrib-left)
    also have ... = 3^n − 3^(n−m+1)
      using sum-geom-nat'[of 3 n−m+1 n] ‹m ≥ 2› ‹m ≤ n›
      by simp
    finally
    have ∑ ?Am' = 3^n − 3^(n−m+1)
      .

    moreover

    have ∑ ?Am'' = 3^(n−m+1)
      by simp

    moreover

    have ∑ ?Am = ∑ ?Am' + ∑ ?Am''
      using Am
      by (simp add: sum.union-disjoint)

    ultimately

    have (∑ ?Am) = (3^n − 3^(n−m+1)) + 3^(n−m+1)
      by simp
    also have ... = 3^n
    proof−
      have (3::nat)^(n−m+1) ≤ 3^n
        using ‹m ≤ n› ‹2 ≤ m›
          by (metis Nat.le-diff-conv2 add.commute add-leD2 diff-diff-cancel
diff-le-self one-le-numeral power-increasing)
      then show ?thesis
        by simp
```

```
  qed
  finally show ?thesis
    .
qed

moreover

have ∑ ?Bm = 3^n
proof−
  have ∑ ?S = 2∗3^n
  proof−
    have ∑ ?Sa = (∑ k∈{1..<n}. 3^k)
    proof−
      have inj-on ((^) (3::nat)) {1..<n}
        unfolding inj-on-def
        by auto
      then show ?thesis
        unfolding Setcompr-eq-image
        by (simp add: sum.reindex-cong)
    qed

    have ∑ ?Sa = (3^n − 1) div 2 − 1
    proof−
      have inj-on (λ k. (3::nat) ^ k) {1..<n}
        unfolding inj-on-def
        by auto
      then have ∑ ?Sa = (∑ k ∈ {1..<n}. 3 ^ k)
        unfolding Setcompr-eq-image
        by (simp add: sum.reindex-cong)
      then show ?thesis
        using sum-geom-nat'[of 3 1 n] ‹n ≥ 3›
        by simp
    qed

    moreover

    have ∑ ?Sb = 2 ∗ ((3^n − 1) div 2 − 1)
    proof−
      have inj-on (λ k. 2 ∗ (3::nat) ^ k) {1..<n}
        unfolding inj-on-def
```

```
      by auto
    then have ∑ ?Sb = (∑ k ∈ {1..<n}. 2 * 3 ^ k)
      unfolding Setcompr-eq-image
      by (simp add: sum.reindex-cong)
    also have ... = 2 * (∑ k ∈ {1..<n}. 3 ^ k)
      by (simp add: sum-distrib-left)
    also have ... = 2 * (∑ ?Sa)
    proof−
      have inj-on (λ k. (3::nat) ^ k) {1..<n}
        unfolding inj-on-def
        by auto
      then show ?thesis
        unfolding Setcompr-eq-image
        by (simp add: sum.reindex-cong)
    qed
    finally
    show ?thesis
      using ⟨∑ ?Sa = (3^n − 1) div 2 − 1⟩
      by simp
  qed

  moreover

  have ∑ ?Sc = (3 ^ n + 9) div 2
    by auto

  moreover

  have ∑ ?S = ∑ ?Sa + ∑ ?Sb + ∑ ?Sc
    using ⟨?Sa ∩ ?Sb = {}⟩ ⟨(?Sa ∪ ?Sb) ∩ ?Sc = {}⟩
    using ⟨finite ?Sa⟩ ⟨finite ?Sb⟩ ⟨finite ?Sc⟩ ⟨finite (?Sa ∪ ?Sb)⟩
    using sum.union-disjoint
    by (metis (no-types, lifting))

  moreover

  have (((3::nat)^n − 1) div 2 − 1) + 2 * ((3^n − 1) div 2 − 1) + (3
^ n + 9) div 2 = 2*3^n (is ?lhs = 2*3^n)
  proof−
    have ((3::nat)^n − 1) div 2 − 1 = (3^n − 3) div 2
```

```
      by simp
    then have ?lhs =  3*((3^n - 3) div 2) + (3 ^ n + 9) div 2
      by simp
    also have ... = ((3*3^n - 9) + (3^n + 9)) div 2
      by (simp add: div-mult-swap)
    also have ... = 2*3^n
    proof−
      have 9 ≤ (3::nat) * 3 ^ n
        using ⟨n ≥ 3⟩
          by (smt Suc-1 ⟨(3 ^ n - 1) div 2 - 1 = (3 ^ n - 3) div 2⟩ calculation diff-add-inverse2 diff-diff-cancel diff-is-0-eq dvd-mult-div-cancel even-add even-power le-add1 le-add-same-cancel2 le-antisym le-trans linear mult-Suc numeral-3-eq-3 odd-two-times-div-two-succ plus-1-eq-Suc power-mult self-le-ge2-pow)
      then have ((3::nat)*3^n - 9) + (3^n + 9) = 4*3^n
        by simp
      then show ?thesis
        by simp
    qed
    finally
    show ?thesis
      .
  qed

  ultimately
  show ?thesis
    by simp
qed
also have ∑ ?S = ∑ ?Am + ∑ ?Bm
  using ⟨?Am ∪ ?Bm = ?S⟩ ⟨?Am ∩ ?Bm = {}⟩ ⟨finite ?Am⟩ ⟨finite ?Bm⟩
  using sum.union-disjoint[of ?Am ?Bm id]
  by simp
then show ?thesis
  using ⟨∑ ?Am = 3^n⟩
  by (metis (no-types, lifting) add-left-cancel calculation mult-2)
qed

ultimately

show ?thesis
  by simp
```

qed

ultimately

show $\exists$ *S1 S2*. *S1* $\cap$ *S2* = {} $\wedge$ *S1* $\cup$ *S2* = *?S* $\wedge$ *card S1* = *m* $\wedge$ $\sum$ *S1* = $\sum$ *S2*
by *blast*
qed
qed

end

5.2.2 IMO 2018 SL - C2

theory *IMO-2018-SL-C2-sol*
imports *Complex-Main*
begin

locale *dim* =
fixes *files* :: *int*
fixes *ranks* :: *int*
assumes *pos*: *files* > *0* $\wedge$ *ranks* > *0*
assumes *div4*: *files mod 4* = *0* $\wedge$ *ranks mod 4* = *0*
begin

type-synonym *square* = *int* $\times$ *int*

definition *squares* :: *square set* **where**
squares = {*0*..<*files*} $\times$ {*0*..<*ranks*}

datatype *piece* = *Queen* | *Knight*

type-synonym *board* = *square* $\Rightarrow$ *piece option*

definition *empty-board* :: *board* **where**
empty-board = (λ *square*. *None*)

fun *attacks-knight* :: *square* $\Rightarrow$ *board* $\Rightarrow$ *bool* **where**
attacks-knight (*file*, *rank*) *board* $\longleftrightarrow$
($\exists$ *file'* *rank'*. (*file'*, *rank'*) $\in$ *squares* $\wedge$ *board* (*file'*, *rank'*) = *Some Knight* $\wedge$

$$((abs\ (file - file') = 1 \wedge abs\ (rank - rank') = 2) \vee (abs\ (file - file') = 2 \wedge abs\ (rank - rank') = 1)))$$

definition *valid-horst-move'* :: *square* ⇒ *board* ⇒ *board* ⇒ *bool* **where**
valid-horst-move' square board board' ⟷
square ∈ *squares* ∧ *board square* = *None* ∧
¬ *attacks-knight square board* ∧
board' = *board* (*square* := *Some Knight*)

definition *valid-horst-move* :: *board* ⇒ *board* ⇒ *bool* **where**
valid-horst-move board board' ⟷
(∃ *square*. *valid-horst-move' square board board'*)

definition *valid-queenie-move* :: *board* ⇒ *board* ⇒ *bool* **where**
valid-queenie-move board board' ⟷
(∃ *square* ∈ *squares*. *board square* = *None* ∧
board' = *board* (*square* := *Some Queen*))

type-synonym *strategy* = *board* ⇒ *board* ⇒ *bool*

inductive *valid-game* :: *strategy* ⇒ *strategy* ⇒ *nat* ⇒ *board* ⇒ *bool* **where**
valid-game horst-strategy queenie-strategy 0 empty-board
| ⟦*valid-game horst-strategy queenie-strategy k board*;
valid-horst-move board board'; *horst-strategy board board'*;
valid-queenie-move board' board''; *queenie-strategy board' board''*⟧⟹ *valid-game horst-strategy queenie-strategy* (*k* + *1*) *board''*

definition *valid-queenie-strategy* :: *strategy* ⇒ *bool* **where**
valid-queenie-strategy queenie-strategy ⟷
(∀ *horst-strategy board board' k*.
valid-game horst-strategy queenie-strategy k board ∧
valid-horst-move board board' ∧ *horst-strategy board board'* ∧
(∃ *square* ∈ *squares*. *board' square* = *None*) ⟶
(∃ *board''*. *valid-queenie-move board' board''* ∧ *queenie-strategy board' board''*))

squares

lemma *squares-card* [*simp*]:
shows *card squares* = *files* ∗ *ranks*
using *pos*

unfolding *squares-def*
by *auto*

lemma *squares-finite* [*simp*]:
shows *finite squares*
using *pos*
unfolding *squares-def*
by *auto*

free-squares

definition *free-squares* :: *board* ⇒ *square set* **where**
free-squares board = {*square* ∈ *squares*. *board square* = *None*}

lemma *free-squares-finite* [*simp*]:
shows *finite* (*free-squares board*)
proof (*rule finite-subset*)
show *free-squares board* ⊆ *squares*
by (*simp add*: *free-squares-def*)
qed *simp*

lemma *valid-game-free-squares-card-even*:
assumes *valid-game horst-strategy queenie-strategy k board*
shows *card* (*free-squares board*) *mod 2* = *0*
using *assms*
proof (*induction horst-strategy queenie-strategy k board rule*: *valid-game.induct*)
case (*1 horst-strategy queenie-strategy*)
show *?case*
proof−
have *card* (*free-squares empty-board*) = *files* ∗ *ranks*
by (*simp add*: *empty-board-def free-squares-def*)
then show *?thesis*
using *div4*
by *presburger*
qed
next
case (*2 horst-strategy queenie-strategy K board board′ board″*)
then obtain *square square′* **where**
square ∈ *squares board square* = *None board′* = *board* (*square* := *Some Knight*)
square′ ∈ *squares board′ square′* = *None board″* = *board′* (*square′* := *Some Queen*)

```
    unfolding valid-horst-move-def valid-horst-move'-def valid-queenie-move-def
    by auto
  then have free-squares board = free-squares board'' ∪ {square, square'}
      square ∉ free-squares board'' square' ∉ free-squares board''
    unfolding free-squares-def
    by (auto split: if-split-asm)
  moreover
  have square ≠ square'
    using ‹board' = board(square ↦ Knight)› ‹board' square' = None›
    by auto
  ultimately
  have card (free-squares board) = card (free-squares board'') + 2
    using card-Un-disjoint[of free-squares board'' {square, square'}]
    by auto
  then show ?case
    using ‹card (free-squares board) mod 2 = 0›
    by simp
qed
```

black squares

```
fun black :: square ⇒ bool where
 black (file, rank) ⟷ (file + rank) mod 2 = 0

definition black-squares :: square set where
  black-squares = {square ∈ squares. black square}

lemma black-squares-finite [simp]:
  shows finite black-squares
  using pos
  unfolding black-squares-def
  by auto

lemma black-squares-card:
  card black-squares = (files * ranks) div 2
proof−
  let ?black-squares = {square ∈ squares. black square}
  let ?white-squares = {square ∈ squares. ¬ black square}
  have squares = ?black-squares ∪ ?white-squares
    by blast
  moreover
```

```
have ?black-squares ∩ ?white-squares = {}
  by blast
moreover
have card ?black-squares = card ?white-squares
proof−
  let ?f = λ (a::int, b::int). if a mod 2 = 0 then (a, b + 1) else (a, b − 1)
  have bij-betw ?f ?black-squares ?white-squares
    unfolding bij-betw-def
  proof
    show inj-on ?f ?black-squares
      unfolding inj-on-def
      by auto
  next
    show ?f ‘ ?black-squares = ?white-squares
    proof
      show ?f ‘ ?black-squares ⊆ ?white-squares
        using div4
        by (auto simp add: squares-def split: if-split-asm) presburger+
    next
      show ?white-squares ⊆ ?f ‘ ?black-squares
      proof
        fix wsq
        assume wsq ∈ ?white-squares
        let ?invf = λ (a, b). if a mod 2 = 0 then (a, b − 1) else (a, b + 1)
        have ?f (?invf wsq) = wsq
          by (cases wsq, auto)
        moreover
        have ?invf wsq ∈ ?black-squares
          using ‹wsq ∈ ?white-squares› div4
          by (cases wsq, auto simp add: squares-def) presburger+
        ultimately
        show wsq ∈ ?f ‘ ?black-squares
          by force
      qed
    qed
  qed
  then show ?thesis
    using bij-betw-same-card by blast
qed
ultimately
```

have *2 ∗ card ?black-squares = card squares*
by (*metis* (*no-types*, *lifting*) *card.infinite card-Un-disjoint finite-Un mult-2 mult-eq-0-iff*)
then have *2 ∗ card ?black-squares = files ∗ ranks*
by *auto*
then show *?thesis*
unfolding *black-squares-def*
by *simp*
qed

free black squares

definition *free-black-squares* :: *board* ⇒ *square set* **where**
free-black-squares board = {*square* ∈ *squares*. *black square* ∧ *board square* = *None*}

lemma *free-black-squares-add-piece*:
shows *card* (*free-black-squares board*) ≤ *card* (*free-black-squares* (*board* (*square* := *Some piece*))) + *1*
proof−
let *?board′* = *board* (*square* := *Some piece*)
have *free-black-squares board* = *free-black-squares ?board′* ∨
free-black-squares board = *free-black-squares ?board′* ∪ {*square*}
unfolding *free-black-squares-def Let-def*
by *auto*
then show *?thesis*
by (*metis One-nat-def add.right-neutral add-Suc-right card.infinite card-Un-le card-empty card-insert-if finite-Un finite-insert insert-absorb insert-not-empty le-add1 trans-le-add2*)
qed

lemma *free-black-squares-valid-horst-move*:
assumes *valid-horst-move board board′*
shows *card* (*free-black-squares board*) ≤ *card* (*free-black-squares board′*) + *1*
using *assms*
using *free-black-squares-add-piece*
unfolding *valid-horst-move-def valid-horst-move′-def free-black-squares-def*
by *auto*

lemma *free-black-squares-valid-queenie-move*:
assumes *valid-queenie-move board board′*

shows *card* (*free-black-squares board*) ≤ *card* (*free-black-squares board'*) + *1*
using *assms*
using *free-black-squares-add-piece*
unfolding *valid-queenie-move-def free-black-squares-def*
by *auto*

knights

definition *knights* :: *board* ⇒ *square set* **where**
knights board = {*square* ∈ *squares*. *board square* = *Some Knight*}

lemma *knights-finite* [*simp*]:
shows *finite* (*knights board*)
by (*rule finite-subset*[*of* - *squares*], *simp-all add*: *knights-def*)

lemma *knights-card-horst-move* [*simp*]:
assumes *valid-horst-move board board'*
shows *card* (*knights board'*) = *card* (*knights board*) + *1*
proof−
obtain *square* **where** *square* ∈ *squares board square* = *None board' square* = *Some Knight*
board' = *board* (*square* := *Some Knight*)
using *assms*
unfolding *valid-horst-move-def valid-horst-move'-def*
by *auto*
then have *knights board'* = *knights board* ∪ {*square*}
unfolding *knights-def*
by *auto*
then show *?thesis*
using ‹*board square* = *None*›
unfolding *knights-def*
by *auto*
qed

lemma *knights-card-queenie-move* [*simp*]:
assumes *valid-queenie-move board board'*
shows *card* (*knights board'*) = *card* (*knights board*)
proof−
have *knights board'* = *knights board*
using *assms*
unfolding *valid-queenie-move-def knights-def*

```
    by force
  then show ?thesis
    by simp
qed

lemma valid-game-knights-card [simp]:
  assumes valid-game horst-strategy queenie-strategy k board
  shows card (knights board) = k
  using assms
proof (induction horst-strategy queenie-strategy k board rule: valid-game.induct)
  case (1 horst-strategy queenie-strategy)
  show ?case
    by (simp add: empty-board-def knights-def)
next
  case (2 horst-strategy queenie-strategy K board board' board'')
  then show ?case
    by auto
qed
```

Cycles

```
fun cycle-opposite :: square ⇒ square where
  cycle-opposite (file, rank) = (4 * (file div 4) + (3 - file mod 4), 4 * (rank div
4) + (3 - rank mod 4))

lemma cycle-opposite-cycle-opposite [simp]:
  shows cycle-opposite (cycle-opposite square) = square
  by (cases square) auto

lemma cycle-opposite-different [simp]:
  shows cycle-opposite square ≠ square
  by (cases square, simp, presburger)

lemma cycle-opposite-squares [simp]:
  shows cycle-opposite square ∈ squares ⟷ square ∈ squares
  using pos div4
  by (cases square) (simp add: squares-def, safe, presburger+)


fun cycle4 :: square ⇒ int where
  cycle4 (x, y) =
```

```
  (if x = 0 then y
   else if x = 1 then (y + 2) mod 4
   else if x = 2 then (5 − y) mod 4
   else 3 − y)

lemma cycle-lt-4:
  assumes 0 ≤ x x < 4 0 ≤ y y < 4
  shows 0 ≤ cycle4 (x, y) ∧ cycle4 (x, y) < 4
  using assms
  by auto

lemma cycle0:
  assumes 0 ≤ x x < 4 0 ≤ y y < 4
  shows cycle4 (x, y) = 0 ⟷ (x, y) ∈ set [(0, 0), (2, 1), (1, 2), (3, 3)]
  using assms
  by auto presburger+

lemma cycle1:
  assumes 0 ≤ x x < 4 0 ≤ y y < 4
  shows cycle4 (x, y) = 1 ⟷ (x, y) ∈ set [(0, 1), (1, 3), (3, 2), (2, 0)]
  using assms
  by auto presburger+

lemma cycle2:
  assumes 0 ≤ x x < 4 0 ≤ y y < 4
  shows cycle4 (x, y) = 2 ⟷ (x, y) ∈ set [(0, 2), (2, 3), (1, 0), (3, 1)]
  using assms
  by auto presburger+

lemma cycle3:
  assumes 0 ≤ x x < 4 0 ≤ y y < 4
  shows cycle4 (x, y) = 3 ⟷ (x, y) ∈ set [(0, 3), (1, 1), (2, 2), (3, 0)]
  using assms
  by auto presburger+

fun cycle :: square ⇒ int × int × int where
  cycle (x, y) = (x div 4, y div 4, cycle4 (x mod 4, y mod 4))

lemma cycles-card:
  shows card (cycle ‘ squares) = (files ∗ ranks) div 4
```

```
proof−
  have cycle ‘ squares = {(x, y, z). x ∈ {0..<files div 4} ∧ y ∈ {0..<ranks div
4} ∧ z ∈ {0..<4}}
  proof safe
    fix f r x y z
    assume (f, r) ∈ squares (x, y, z) = cycle (f, r)
    then have 0 ≤ f ∧ f < files 0 ≤ r ∧ r < ranks
      by (auto simp add: squares-def)
    then have 0 ≤ f div 4 ∧ f div 4 < files div 4  0 ≤ r div 4 ∧ r div 4 < ranks
div 4
      using div4
      by presburger+
    then show x ∈ {0..<files div 4} y ∈ {0..<ranks div 4}
      using ‹(x, y, z) = cycle (f, r)›
      by auto
    show z ∈ {0..<4}
      using cycle-lt-4[rule-format, of f mod 4 r mod 4]
      using ‹(x, y, z) = cycle (f, r)›
      by simp
  next
    fix x y z :: int
    assume ∗: x ∈ {0..<files div 4} y ∈ {0..<ranks div 4} z ∈ {0..<4}
    let ?f = 4 ∗ x and ?r = 4 ∗ y + z
    have (?f, ?r) ∈ squares cycle (?f, ?r) = (x, y, z)
      using ∗
      by (auto simp add: squares-def)
    then have ∃ square ∈ squares. cycle square = (x, y, z)
      by blast
    then show (x, y, z) ∈ cycle ‘ squares
      by (metis imageI)
  qed
  also have ... = {0..<files div 4} × {0..<ranks div 4} × {0..<4}
    by auto
  finally
  have card (cycle ‘ squares) = (files div 4) ∗ (ranks div 4) ∗ 4
    using pos
    by simp
  also have ... = (files ∗ ranks) div 4
    using div4
    by auto
```

```
  finally show ?thesis
    .
qed

lemma cycle4-exhausted:
  assumes 0 ≤ f1 f1 < 4 0 ≤ r1 r1 < 4
  assumes 0 ≤ f2 f2 < 4 0 ≤ r2 r2 < 4
  assumes (f1, r1) ≠ (f2, r2)
          abs (f1 − f2) ≠ 1 ∨ abs (r1 − r2) ≠ 2
          abs (f1 − f2) ≠ 2 ∨ abs (r1 − r2) ≠ 1
          (f2, r2) ≠ (3 − f1, 3 − r1)
  shows cycle4 (f1, r1) ≠ cycle4 (f2, r2)
  using assms cycle-lt-4[rule-format, of f1 r1]
  by (smt cycle0 cycle1 cycle2 cycle3 list.set-intros(1) list.set-intros(2))

lemma cycle-exhausted:
  assumes ∀ sq ∈ squares. board sq = Some Knight ⟶ ¬ attacks-knight sq board
          ∀ sq ∈ squares. board sq = Some Knight ⟶ board (cycle-opposite sq) =
Some Queen
          sq1 ≠ sq2 sq1 ∈ squares sq2 ∈ squares board sq1 = Some Knight board
sq2 = Some Knight
  shows cycle sq1 ≠ cycle sq2
proof safe
  assume cycle sq1 = cycle sq2
  obtain f1 r1 where sq1: sq1 = (f1, r1)
    by (cases sq1)
  obtain f2 r2 where sq2: sq2 = (f2, r2)
    by (cases sq2)

  have **: f1 div 4 = f2 div 4 r1 div 4 = r2 div 4
           cycle4 (f1 mod 4, r1 mod 4) = cycle4 (f2 mod 4, r2 mod 4)
    using ‹cycle sq1 = cycle sq2› sq1 sq2
    by simp-all

  have ¬ attacks-knight (f1, r1) board (f2, r2) ≠ cycle-opposite (f1, r1)
    using assms(1)[rule-format, of (f1, r1)]
    using assms(2)[rule-format, of (f1, r1)]
    using assms(4−7) sq1 sq2
    by auto
```

have *f2* ≠ *4* ∗ (*f1 div 4*) + (*3* − *f1 mod 4*) ∨ *r2* ≠ *4* ∗ (*r1 div 4*) + (*3* − *r1 mod 4*)
using ⟨(*f2*, *r2*) ≠ *cycle-opposite* (*f1*, *r1*)⟩
by *auto*

then have *f2 mod 4* ≠ *3* − *f1 mod 4* ∨ *r2 mod 4* ≠ *3* − *r1 mod 4*
using ∗∗(*1*−*2*)
by *safe presburger*+

then have *1*: (*f2 mod 4*, *r2 mod 4*) ≠ (*3* − *f1 mod 4*, *3* − *r1 mod 4*)
by *simp*

have (|*f1* − *f2*| = *1* ⟶ |*r1* − *r2*| ≠ *2*) ∧ (|*f1* − *f2*| = *2* ⟶ |*r1* − *r2*| ≠ *1*)
using ⟨¬ *attacks-knight* (*f1*, *r1*) *board*⟩
using *assms attacks-knight.simps sq1 sq2*
by *blast*

then have *2*: |*f1 mod 4* − *f2 mod 4*| ≠ *1* ∨ |*r1 mod 4* − *r2 mod 4*| ≠ *2*
|*f1 mod 4* − *f2 mod 4*| ≠ *2* ∨ |*r1 mod 4* − *r2 mod 4*| ≠ *1*
using ∗∗(*1*−*2*)
by (*smt mult-div-mod-eq*)+

have (*f1 mod 4*, *r1 mod 4*) = (*f2 mod 4*, *r2 mod 4*)
using ∗∗(*3*) *cycle4-exhausted*[*OF* - - - - - - - - - *2 1*]
using *pos-mod-conj zero-less-numeral*
by *blast*

then have *f1* = *f2 r1* = *r2*
using ∗∗(*1*−*2*)
by (*metis mult-div-mod-eq prod.inject*)+

then show *False*
using *sq1 sq2* ⟨*sq1* ≠ *sq2*⟩
by *simp*
qed

guaranteed game lengths

definition *guaranteed-game-lengths* :: *nat set* **where**
guaranteed-game-lengths = {*K*. ∃ *horst-strategy*. ∀ *queenie-strategy*. *valid-queenie-strategy queenie-strategy* ⟶ (∃ *board*. *valid-game horst-strategy queenie-strategy K board*)}

lemma *guaranteed-game-lengths-geq*:
shows *nat* ((*files* ∗ *ranks*) *div* 4) ∈ *guaranteed-game-lengths*
unfolding *guaranteed-game-lengths-def*
proof *safe*
let *?l* = *nat* ((*files* ∗ *ranks*) *div* 4)
let *?horst-strategy* = λ *board board′*:: *board*. (∃ *square*. *black square* ∧ *valid-horst-move′ square board board′*)
show ∃ *horst-strategy*. ∀ *queenie-strategy*. *valid-queenie-strategy queenie-strategy* ⟶ (∃ *board*. *valid-game horst-strategy queenie-strategy ?l board*)
proof (*rule-tac x*=*?horst-strategy* **in** *exI*, *safe*)
fix *queenie-strategy*
assume *valid-queenie-strategy queenie-strategy*

have *1*: ∀ *k board*. *valid-game ?horst-strategy queenie-strategy k board* ⟶ (∀ *square* ∈ *squares*. *board square* = *Some Knight* ⟶ *black square*) (**is** ∀ *k*. *?P k*)
proof *safe*
fix *k board f r*
assume *valid-game ?horst-strategy queenie-strategy k board*
(*f*, *r*) ∈ *squares board* (*f*, *r*) = *Some Knight*
then show *black* (*f*, *r*)
proof (*induction ?horst-strategy queenie-strategy k board rule*: *valid-game.induct*)
case (*1 queenie-strategy*)
then show *?case*
by (*simp add*: *empty-board-def*)
next
case (*2 queenie-strategy K board board′ board″*)
then show *?case*
by (*smt map-upd-Some-unfold piece.simps*(*1*) *valid-horst-move′-def valid-queenie-move-def*)
qed
qed

have ∀ *k* ≤ (*files* ∗ *ranks*) *div* 4. ∃ *board*. *valid-game ?horst-strategy queenie-strategy k board*
proof *safe*
fix *k*::*nat*
assume *k* ≤ (*files* ∗ *ranks*) *div* 4
then show ∃ *board*. *valid-game ?horst-strategy queenie-strategy k board*
proof (*induction k*)

```
    case 0
    then show ?case
      by (rule-tac x=empty-board in exI, simp add: valid-game.intros)
  next
    case (Suc k)
      then obtain board where valid-game ?horst-strategy queenie-strategy k
board
      by auto
    then have *: (files * ranks) div 2 - 2 * k ≤ card (free-black-squares board)
      using ‹Suc k ≤ (files * ranks) div 4›
    proof (induction ?horst-strategy queenie-strategy k board rule: valid-game.induct)
      case 1
      then show ?case
        using black-squares-card
        by (simp add: empty-board-def black-squares-def free-black-squares-def)
    next
      case (2 queenie-strategy k board board' board'')
      then have (files * ranks) div 2 - 2 * k ≤ card (free-black-squares board)
        by auto
      also have ... ≤ card (free-black-squares board') + 1
        using 2
        using free-black-squares-valid-horst-move[of board board']
        by simp
      also have ... ≤ card (free-black-squares board'') + 2
        using 2
        using free-black-squares-valid-queenie-move[of board' board'']
        by simp
      finally show ?case
        using ‹Suc (k + 1) ≤ (files * ranks) div 4›
        by (simp add: le-diff-conv)
    qed
    then have card (free-black-squares board) > 0
      using ‹Suc k ≤ (files * ranks) div 4›
      by auto
    then obtain square where square ∈ free-black-squares board
    by (metis Collect-empty-eq Collect-mem-eq card.infinite card-0-eq not-less0)

    have ¬ attacks-knight square board
    proof (rule ccontr)
      obtain x y where square = (x, y)
```

by (*cases square*)
assume ¬ *?thesis*
then obtain x' y' **where** $(x', y') \in$ *squares board* (x', y') = *Some Knight* $|x - x'| = 1 \wedge |y - y'| = 2 \vee |x - x'| = 2 \wedge |y - y'| = 1$
using ‹*square* = (x, y)›
by *auto*
then have *black* (x', y')
using *1*[*rule-format*, *OF* ‹*valid-game ?horst-strategy queenie-strategy k board*›]
by *auto*

have *black* (x, y)
using ‹*square* ∈ *free-black-squares board*› ‹*square* = (x, y)›
by (*simp add*: *free-black-squares-def*)

show *False*
using ‹*black* (x, y)› ‹*black* (x', y')› ‹$|x - x'| = 1 \wedge |y - y'| = 2 \vee |x - x'| = 2 \wedge |y - y'| = 1$›
unfolding *black.simps*
by *presburger*
qed

let *?board1* = *board* (*square* := *Some Knight*)
have *valid-horst-move board ?board1*
using ‹*square* ∈ *free-black-squares board*› ‹¬ *attacks-knight square board*›
unfolding *valid-horst-move-def valid-horst-move'-def*
by (*rule-tac x*=*square* **in** *exI*, *cases square*, *simp add*: *free-black-squares-def*)

moreover

have *?horst-strategy board ?board1*
using ‹*valid-horst-move board ?board1*› ‹*square* ∈ *free-black-squares board*›
unfolding *valid-horst-move-def free-black-squares-def*
by (*rule-tac x*=*square* **in** *exI*, *cases square*)
(*metis* (*mono-tags*, *lifting*) *map-upd-Some-unfold mem-Collect-eq option.discI valid-horst-move'-def*)

moreover

have ∃ *square* ∈ *squares*. *?board1 square* = *None*

```
      proof−
        have card (free-squares board) mod 2 = 0
          using ⟨valid-game ?horst-strategy queenie-strategy k board⟩
          using valid-game-free-squares-card-even
          by blast
           have free-squares board = free-squares ?board1 ∪ {square} square ∉
free-squares ?board1
          using ⟨square ∈ free-black-squares board⟩
          unfolding free-black-squares-def free-squares-def
          by auto
        then have card (free-squares board) = card (free-squares ?board1) + 1
          by auto
        then have card (free-squares ?board1) mod 2 = 1
          using ⟨card (free-squares board) mod 2 = 0⟩
          by presburger
        then have free-squares ?board1 ≠ {}
          by auto
        then show ?thesis
          unfolding free-squares-def
          by blast
      qed

    then obtain board2 where valid-queenie-move ?board1 board2 queenie-strategy
?board1 board2
        using ⟨valid-queenie-strategy queenie-strategy⟩
        unfolding valid-queenie-strategy-def
        using ⟨valid-game ?horst-strategy queenie-strategy k board⟩ calculation(1)
calculation(2) valid-horst-move'-def
        by blast

      ultimately

      show ?case
        using ⟨valid-game ?horst-strategy queenie-strategy k board⟩
        by (metis (no-types, lifting) Suc-eq-plus1  valid-game.intros(2))
    qed
  qed
  then show ∃ board. valid-game ?horst-strategy queenie-strategy ?l board
    using pos
    by simp
```

```
  qed
qed

lemma valid-game-not-attacks-knight:
  assumes valid-game horst-strategy queenie-strategy k board
          square ∈ squares board square = Some Knight
        shows ¬ attacks-knight square board
  using assms
proof (induction horst-strategy queenie-strategy k board rule: valid-game.induct)
  case (1 horst-strategy queenie-strategy)
  then show ?case
    by (simp add: empty-board-def)
next
  case (2 horst-strategy queenie-strategy K board board' board'')
  have ¬ attacks-knight square board'
  proof (cases board square = Some Knight)
    case True
    then have ¬ attacks-knight square board
      using 2
      by simp
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      obtain x y where square = (x, y)
        by (cases square)
      then obtain x' y' where (x', y') ∈ squares board' (x', y') = Some Knight
        |x − x'| = 1 ∧ |y − y'| = 2 ∨ |x − x'| = 2 ∧ |y − y'| = 1
        using ‹¬ ¬ attacks-knight square board'›
        by auto
      obtain square' where
        square' ∈ squares ¬ attacks-knight square' board
        board square' = None board' = board (square' := Some Knight)
        using ‹valid-horst-move board board'›
        unfolding valid-horst-move-def valid-horst-move'-def
        by auto
      have square' = (x', y')
        using ‹|x − x'| = 1 ∧ |y − y'| = 2 ∨ |x − x'| = 2 ∧ |y − y'| = 1›
       using ‹¬ attacks-knight square board› ‹board' (x', y') = Some Knight› ‹board'
= board(square' ↦ Knight)› ‹(x', y') ∈ squares› ‹square = (x, y)›
        by (metis (full-types)  attacks-knight.simps fun-upd-other)
```

```
      then have attacks-knight square' board
        using ⟨square' ∈ squares⟩ ⟨|x − x'| = 1 ∧ |y − y'| = 2 ∨ |x − x'| = 2 ∧
|y − y'| = 1⟩
              ⟨board square = Some Knight⟩ ⟨square = (x, y)⟩
        using ⟨square ∈ squares⟩ ⟨board square = Some Knight⟩
        by (smt attacks-knight.simps)
      then show False
        using ⟨¬ attacks-knight square' board⟩
        by simp
    qed
  next
    case False
    have board' square = Some Knight
      using ⟨square ∈ squares⟩ ⟨board'' square = Some Knight⟩ ⟨valid-queenie-move
board' board''⟩
      by (metis map-upd-Some-unfold piece.distinct(1) valid-queenie-move-def)

    obtain square' where ∗: square' ∈ squares
      board square' = None ¬ attacks-knight square' board
      board' = board(square' ↦ Knight)
      using ⟨valid-horst-move board board'⟩
      unfolding valid-horst-move-def valid-horst-move'-def
      by blast
    then have square = square'
      using ⟨board square ≠ Some Knight⟩
      using ⟨board' square = Some Knight⟩
      by (metis fun-upd-apply)
    then have ¬ attacks-knight square board
      using ⟨¬ attacks-knight square' board⟩
      by simp
    then show ?thesis
      by (cases square) (simp add: ∗(4) ⟨square = square'⟩)
  qed
  then show ?case
    using ⟨valid-queenie-move board' board''⟩
   by (smt attacks-knight.elims(2) attacks-knight.elims(3) fun-upd-apply option.inject
piece.simps(1) prod.simps(1) valid-queenie-move-def)
qed

lemma guaranteed-game-lengths-leq:
```

```
  shows ∀ k ∈ guaranteed-game-lengths. k ≤ (files * ranks) div 4
proof safe
  fix k
  assume k ∈ guaranteed-game-lengths
  then obtain horst-strategy where
    *: ∀ queenie-strategy. valid-queenie-strategy queenie-strategy ⟶
                        (∃ board. valid-game horst-strategy queenie-strategy k board)
    unfolding guaranteed-game-lengths-def
    by auto
  show k ≤ (files * ranks) div 4
  proof (rule ccontr)
    assume ¬ ?thesis
    then have k > (files * ranks) div 4
      by simp

     let ?queenie-strategy = λ board board'. (∃ square ∈ squares. board square
= Some Knight ∧ board (cycle-opposite square) = None ∧ board' (cycle-opposite
square) = Some Queen)

    have 1: ∀ k horst-strategy board. valid-game horst-strategy ?queenie-strategy k
board ⟶
                  (∀ square ∈ squares. board square = Some Knight ⟷ board
(cycle-opposite square) = Some Queen) (is ∀ k. ?P k)
    proof (rule allI, rule allI, rule allI, rule impI, rule ballI)
      fix k horst-strategy board square
      assume valid-game horst-strategy ?queenie-strategy k board square ∈ squares
      then show (board square = Some Knight) = (board (cycle-opposite square)
= Some Queen)
       proof (induction horst-strategy ?queenie-strategy k board arbitrary: square
rule: valid-game.induct)
        case (1 horst-strategy)
        then show ?case
          by (simp add: empty-board-def)
      next
        case (2 horst-strategy K board board' board'')
        show ?case
        proof safe
          assume board'' square = Some Knight
          show board'' (cycle-opposite square) = Some Queen
          proof (cases board square = Some Knight)
```

```
      case True
      then have board (cycle-opposite square) = Some Queen
        using 2
        by blast
      then have board' (cycle-opposite square) = Some Queen
        using ⟨valid-horst-move board board'⟩
        unfolding valid-horst-move-def valid-horst-move'-def
        by (metis fun-upd-apply option.distinct(1))
      then show ?thesis
        using ⟨valid-queenie-move board' board''⟩
        using valid-queenie-move-def
        by auto
    next
      case False
     from ⟨valid-queenie-move board' board''⟩ ⟨?queenie-strategy board' board''⟩
      obtain square' where
        square' ∈ squares
        board' square' = Some Knight
        board' (cycle-opposite square') = None
        board'' (cycle-opposite square') = Some Queen
        by auto

      have square = square'
      proof (rule ccontr)
        assume square ≠ square'
        then have board square' = Some Knight
         using ⟨board'' square = Some Knight⟩ ⟨board' square' = Some Knight⟩
⟨valid-horst-move board board'⟩ ⟨valid-queenie-move board' board''⟩
       by (smt False map-upd-Some-unfold piece.distinct(1) valid-horst-move'-def
valid-horst-move-def valid-queenie-move-def)
        then have board (cycle-opposite square') = Some Queen
          using ⟨square' ∈ squares⟩ 2
          by simp
        then have board' (cycle-opposite square') = Some Queen
          by (metis ⟨board' (cycle-opposite square') = None⟩ ⟨valid-horst-move
board board'⟩ fun-upd-def valid-horst-move'-def valid-horst-move-def)
        then show False
          using ⟨board' (cycle-opposite square') = None⟩
          by simp
      qed
```

```
    then show ?thesis
      using ⟨board'' (cycle-opposite square') = Some Queen⟩
      by simp
  qed
next
  assume board'' (cycle-opposite square) = Some Queen
  show board'' square = Some Knight
  proof (cases board (cycle-opposite square) = Some Queen)
    case True
    then have board square = Some Knight
      using 2
      by auto
    then have board' square = Some Knight
      using ⟨valid-horst-move board board'⟩
  unfolding valid-horst-move-def valid-horst-move'-def valid-queenie-move-def
      by auto
    then show ?thesis
      using ⟨valid-queenie-move board' board''⟩
      unfolding valid-queenie-move-def
      by auto
  next
    case False
    then have board' (cycle-opposite square) ≠ Some Queen
      using ⟨valid-horst-move board board'⟩
  unfolding valid-horst-move-def valid-horst-move'-def valid-queenie-move-def
      by (meson map-upd-Some-unfold piece.simps(2))
    obtain square' where square' ∈ squares
      board' (cycle-opposite square') = None
      board'' (cycle-opposite square') = Some Queen
      board' square' = Some Knight
      using ⟨?queenie-strategy board' board''⟩
      by auto
    moreover
    obtain square'' where board' square'' = None
      board'' = board' (square'' := Some Queen)
      using ⟨valid-queenie-move board' board''⟩
      unfolding valid-queenie-move-def
      by auto
    ultimately
    have cycle-opposite square' = square''
```

```
          by (auto split: if-split-asm)
        then have cycle-opposite square' = cycle-opposite square
          using ⟨board'' (cycle-opposite square) = Some Queen⟩
          using ⟨board' (cycle-opposite square) ≠ Some Queen⟩
          using ⟨board'' = board' (square'' := Some Queen)⟩
          by (auto split: if-split-asm)
            then have cycle-opposite (cycle-opposite square') = cycle-opposite
(cycle-opposite square)
          by simp
        then have square' = square
          by simp
        then have board' square = Some Knight
          using ⟨board' square' = Some Knight⟩
          by simp
        then show ?thesis
          using ⟨board'' = board'(square'' ↦ Queen)⟩
                ⟨board' (cycle-opposite square') = None⟩
                ⟨cycle-opposite square' = square''⟩ ⟨square' = square⟩
          by auto
      qed
    qed
  qed
qed

have valid-queenie-strategy ?queenie-strategy
  unfolding valid-queenie-strategy-def
proof safe
  fix horst-strategy board board' k f r
  assume valid-game horst-strategy ?queenie-strategy k board
         valid-horst-move board board' horst-strategy board board'
  then obtain square where
  ∗: square ∈ squares board square = None ¬ attacks-knight square board board'
= board(square ↦ Knight)
    unfolding valid-horst-move-def valid-horst-move'-def
    by auto
     have board (cycle-opposite square) ≠ Some Queen board (cycle-opposite
square) ≠ Some Knight
       using 1[rule-format, OF ⟨valid-game horst-strategy ?queenie-strategy k
board⟩, of square]
       using 1[rule-format, OF ⟨valid-game horst-strategy ?queenie-strategy k
```

board⟩, *of cycle-opposite square*]
using ⟨*square* ∈ *squares*⟩ ⟨*board square* = *None*⟩
by *auto*
then have *board* (*cycle-opposite square*) = *None*
by (*metis* (*full-types*) *option.exhaust-sel piece.exhaust*)

let *?board* = *board'* (*cycle-opposite square* := *Some Queen*)
have *?queenie-strategy board' ?board*
using ∗ ⟨*board* (*cycle-opposite square*) = *None*⟩ ⟨*square* ∈ *squares*⟩
by (*rule-tac x*=*square* **in** *bexI*, *simp-all*)

moreover

obtain *f' r'* **where** *cycle-opposite square* = (*f'*, *r'*)
by (*cases cycle-opposite square*)
then have *valid-queenie-move board' ?board*
using ⟨*board* (*cycle-opposite square*) = *None*⟩ *cycle-opposite-squares*[*of square*]
unfolding *valid-queenie-move-def*
by (*metis* ∗(*1*) ∗(*4*) *cycle-opposite-different fun-upd-other*)

ultimately
show ∃ *board''*.
valid-queenie-move board' board'' ∧
?queenie-strategy board' board''
by *blast*
qed

then obtain *board* **where** ∗∗: *valid-game horst-strategy ?queenie-strategy k board*
using ∗
by *auto*

have *card* (*knights board*) > (*files* ∗ *ranks*) *div 4*
using *valid-game-knights-card*[*rule-format*, *OF* ∗∗] ⟨*k* > (*files* ∗ *ranks*) *div 4*⟩
by *auto*

have *card* (*cycle* ‘ (*knights board*)) > (*files* ∗ *ranks*) *div 4*
proof−
have *inj-on cycle* (*knights board*)

```
      unfolding inj-on-def
    proof (rule ballI, rule ballI, rule impI)
      fix square1 square2
      assume square1 ∈ knights board square2 ∈ knights board cycle square1 =
cycle square2
      then show square1 = square2
        using 1[rule-format, OF ‹valid-game horst-strategy ?queenie-strategy k
board›]
      using valid-game-not-attacks-knight[rule-format, OF ‹valid-game horst-strategy
?queenie-strategy k board›]
        using cycle-exhausted[of board]
        unfolding knights-def
        by blast
    qed
    then show ?thesis
      using ‹card (knights board) > (files * ranks) div 4›
      by (simp add: card-image)
  qed

  moreover

  have cycle ‘ (knights board) ⊆ cycle ‘ squares
    unfolding knights-def
    by auto

  moreover

  have finite (cycle ‘ squares)
    by simp

  ultimately

  have card (cycle ‘ squares) > (files * ranks) div 4
    using card-mono
    by (smt zle-int)

  then show False
    using cycles-card
    by simp
qed
```

```
qed

lemma guaranteed-game-lengths-finite:
  shows finite guaranteed-game-lengths
proof (subst finite-nat-set-iff-bounded-le)
  show ∃ m. ∀ n∈guaranteed-game-lengths. n ≤ m
  proof (rule-tac x=nat ((files*ranks) div 4) in exI)
    show ∀ n∈guaranteed-game-lengths. n ≤ nat (files * ranks div 4)
      using guaranteed-game-lengths-leq pos
      by auto
  qed
qed

theorem Max guaranteed-game-lengths = nat ((files * ranks) div 4)
proof (rule Max-eqI)
  show nat ((files * ranks) div 4) ∈ guaranteed-game-lengths
    using guaranteed-game-lengths-geq
    by auto
next
  fix k
  assume k ∈ guaranteed-game-lengths
  then show k ≤ nat ((files * ranks) div 4)
    using guaranteed-game-lengths-leq
    by auto
next
  show finite guaranteed-game-lengths
    using guaranteed-game-lengths-finite
    by auto
qed

end

end
```

5.2.3 IMO 2018 SL - C3

```
theory IMO-2018-SL-C3-sol
  imports Complex-Main
begin
```

General lemmas

lemma *sum-list-int* [*simp*]:
fixes *xs* :: *nat list*
shows $(\sum\ x \leftarrow xs.\ int\ (f\ x)) = int\ (\sum\ x \leftarrow xs.\ f\ x)$
by (*induction xs*, *auto*)

lemma *sum-list-comp*:
shows $(\sum\ x \leftarrow xs.\ f\ (g\ x)) = (\sum\ x \leftarrow map\ g\ xs.\ f\ x)$
by (*induction xs*, *auto*)

lemma *lt-ceiling-frac*:
assumes $x < ceiling\ (a\ /\ b)\ b > 0$
shows $x * b < a$
using *assms*
by (*metis* (*no-types*, *hide-lams*) *floor-less-iff floor-uminus-of-int less-ceiling-iff minus-mult-minus mult-minus-right of-int-0-less-iff of-int-minus of-int-mult pos-less-divide-eq*)

lemma *subset-Max*:
fixes *X* :: *nat set*
assumes *finite X*
shows $X \subseteq \{0..<Max\ X + 1\}$
using *assms*
by (*induction X rule*: *finite.induct*) (*auto simp add*: *less-Suc-eq-le subsetI*)

lemma *card-Max*:
fixes *X* :: *nat set*
shows $card\ X \leq Max\ X + 1$
proof (*cases finite X*)
case *True*
then show *?thesis*
using *subset-Max*[*of X*]
using *subset-eq-atLeast0-lessThan-card* **by** *blast*
next
case *False*
then show *?thesis*
by *simp*
qed

lemma *sum-length-parts*:

```
  assumes ∀ i j. i < j ∧ j < length ps ⟶ set (filter (ps ! i) xs) ∩ set (filter (ps
! j) xs) = {}
  shows sum-list (map (λ p. length (filter p xs)) ps) ≤ length xs
  using assms
proof (induction ps arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ps)
  let ?xs' = filter (λ x. ¬ p x) xs
  have (∑ p←ps. length (filter p xs)) = (∑ p←ps. length (filter p ?xs'))
  proof−
    have ∗: ∀ p' ∈ set ps. set (filter p xs) ∩ set (filter p' xs) = {}
      using Cons(2)[rule-format, of 0]
     by (metis Suc-less-eq in-set-conv-nth length-Cons list.sel(3) nth-Cons-0 nth-tl
zero-less-Suc)
    have ∀ p ∈ set ps. filter p xs = filter p ?xs'
    proof
      fix p'
      assume p' ∈ set ps
      then have set (filter p xs) ∩ set (filter p' xs) = {}
        using ∗
        by auto
      show filter p' xs = filter p' ?xs'
      proof (subst filter-filter, rule filter-cong)
        fix x
        assume x ∈ set xs
        then show p' x = (¬ p x ∧ p' x)
          using ⟨set (filter p xs) ∩ set (filter p' xs) = {}⟩
          by auto
      qed simp
    qed
    then have ∀ p ∈ set ps. length (filter p xs) = length (filter p ?xs')
      by simp
    then show ?thesis
      by (metis (no-types, lifting) map-eq-conv)
  qed
  moreover
  have (∑ pa←ps. length (filter pa (filter (λx. ¬ p x) xs))) ≤ length (filter (λx.
```

```
¬ p x) xs)
  proof (rule Cons(1), safe)
    fix i j x
    assume i < j j < length ps x ∈ set (filter (ps ! i) ?xs')  x ∈ set (filter (ps !
j) ?xs')
    then have False
      using Cons(2)[rule-format, of i+1 j+1]
      by auto
    then show x ∈ {}
      by simp
  qed

  moreover
  have length (filter p xs) + length (filter (λ x. ¬ p x) xs) = length xs
    using sum-length-filter-compl
    by blast

  ultimately

  show ?case
    by simp
qed


lemma hd-filter:
  assumes filter P xs ≠ []
  shows ∃ k. k < length xs ∧ (filter P xs) ! 0 = xs ! k ∧ P (xs ! k) ∧ (∀ k' < k.
¬ P (xs ! k'))
  using assms
proof (induction xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  show ?case
  proof (cases P x)
    case True
    then show ?thesis
      by auto
```

```
  next
    case False
    then obtain k where k<length xs filter P xs ! 0 = xs ! k P (xs ! k) (∀ k'<k.
¬ P (xs ! k'))
      using Cons
      by auto
    then show ?thesis
      using False
      by (rule-tac x=k+1 in exI, simp add: nth-Cons')
  qed
qed

lemma last-filter:
  assumes filter P xs ≠ []
  shows ∃ k. k < length xs ∧ (filter P xs) ! (length (filter P xs) - 1) = xs ! k ∧
P (xs ! k) ∧ (∀ k'. k < k' ∧ k' < length xs ⟶ ¬ P (xs ! k'))
proof−
  have filter P (rev xs) ≠ []
    using assms
    by (metis Nil-is-rev-conv rev-filter)
  then obtain k where *: k < length xs filter P (rev xs) ! 0 = rev xs ! k P (rev
xs ! k) ∀ k' < k. ¬ P (rev xs ! k')
    using hd-filter[of P rev xs]
    by auto
  show ?thesis
  proof (rule-tac x=length xs - (k + 1) in exI, safe)
    show length xs - (k + 1) < length xs
      using *(1)
      by simp
  next
    show filter P xs ! (length (filter P xs) - 1) = xs ! (length xs - (k + 1))
      using *(1) *(2)
    by (metis One-nat-def add.right-neutral add-Suc-right assms length-greater-0-conv
rev-filter rev-nth)
  next
    show P (xs ! (length xs - (k + 1)))
      using *(1) *(3)
      by (simp add: rev-nth)
  next
    fix k'
```

```
    assume length xs − (k + 1) < k' k' < length xs P (xs ! k')
    then show False
      using ∗(1) ∗(4)[rule-format, of length xs − (k' + 1)]
    by (smt add.commute add-diff-cancel-right add-diff-cancel-right' add-diff-inverse-nat
add-gr-0 diff-less diff-less-mono2 not-less-eq plus-1-eq-Suc rev-nth zero-less-one)
  qed
qed

lemma filter-tl [simp]:
  filter P (tl xs) = (if P (hd xs) then tl (filter P xs) else filter P xs)
   by (smt filter.simps(1) filter.simps(2) filter-empty-conv hd-Cons-tl hd-in-set
list.inject list.sel(2))

lemma filter-dropWhile-not [simp]:
  shows filter P (dropWhile (λx. ¬ P x) xs) = filter P xs
 by (metis (no-types, lifting) filter-False filter-append self-append-conv2 set-takeWhileD
takeWhile-dropWhile-id)

lemma inside-filter:
  assumes i + 1 < length (filter P xs)
  shows ∃ k1 k2. k1 < k2 ∧ k2 < length xs ∧
                 (filter P xs) ! i = xs ! k1 ∧
                 (filter P xs) ! (i + 1) = xs ! k2 ∧
                 P (xs ! k1) ∧ P (xs ! k2) ∧
                 (∀ k'. k1 < k' ∧ k' < k2 ⟶ ¬ P (xs ! k'))
  using assms
proof (induction i arbitrary: xs)
  case 0
  then obtain k1 where k1 < length xs filter P xs ! 0 = xs ! k1 P (xs ! k1) ∀
k' < k1. ¬ P (xs ! k')
    using hd-filter
    by (metis gr-implies-not-zero length-0-conv)
  let ?xs = drop (k1 + 1) xs
  have filter P (take (k1 + 1) xs) = [xs ! k1]
  proof−
    have filter P (take k1 xs) = []
      using ⟨∀ k' < k1. ¬ P (xs ! k')⟩ ⟨k1 < length xs⟩
      using last-filter
      by force
    moreover
```

```
    have take (k1 + 1) xs = take k1 xs @ [xs ! k1]
      using ⟨k1 < length xs⟩
      using take-Suc-conv-app-nth
      by auto
    ultimately
    show ?thesis
      using ⟨P (xs ! k1)⟩
      by simp
  qed
  then have filter P ?xs ≠ []
    using 0
   by (metis One-nat-def Suc-eq-plus1 append-take-drop-id filter-append length-Cons
length-append less-not-refl3 list.size(3) plus-1-eq-Suc)
  then obtain k2' where ∗: k2' < length ?xs filter P ?xs ! 0 = ?xs ! k2' P (?xs
! k2') ∀ k' < k2'. ¬ P (?xs ! k')
    using hd-filter[of P ?xs]
    by auto
  have filter P xs ! 1 = xs ! (k1 + 1 + k2')
    using ∗ ⟨filter P (take (k1 + 1) xs) = [xs ! k1]⟩ ⟨k1 < length xs⟩
     by (metis One-nat-def Suc-eq-plus1 Suc-leI append-take-drop-id filter-append
length-Cons list.size(3) nth-append-length-plus nth-drop plus-1-eq-Suc)
  moreover
  have P (xs ! (k1 + 1 + k2'))
    using ∗ ⟨k1 < length xs⟩
    by auto
  moreover
  have ∀ k'. k1 < k' ∧ k' < k1 + 1 + k2' ⟶ ¬ P (xs ! k')
  proof safe
    fix k'
    assume k1 < k' k' < k1 + 1 + k2' P (xs ! k')
    then have k' − (k1 + 1) < k2'
      by auto
    then have ¬ P (?xs ! (k' − (k1 + 1)))
      using ⟨∀ k' < k2'. ¬ P (?xs ! k')⟩
      by simp
    then have ¬ P (xs ! k')
      using ⟨k2' < length ?xs⟩
      using ⟨k1 < k'⟩
      by auto
    then show False
```

```
    using ‹P (xs ! k')›
    by simp
 qed
 moreover
 have k1 + 1 + k2' < length xs
  using ‹k2' < length ?xs›
  by auto
 ultimately
 show ?case
  using ‹P (xs ! k1)› ‹filter P xs ! 0 = xs ! k1›
  by (rule-tac x=k1 in exI, rule-tac x=k1+1+k2' in exI, simp)
next
 case (Suc i)
 let ?t = takeWhile (λ x. ¬ P x) xs and ?d = dropWhile (λ x. ¬ P x) xs
 let ?xs = tl ?d

 have ?xs ≠ []
  using Suc(2)
  by (metis Suc-eq-plus1 add.commute add-less-cancel-left filter.simps(1) filter-dropWhile-not
filter-tl hd-Cons-tl length-Cons list.size(3) not-less-zero)

 have ∗: ∀ k. length ?t + k + 1 < length xs ⟶ xs ! (length ?t + k + 1) = tl
?d ! k
   by (metis One-nat-def add.right-neutral add-Suc-right add-lessD1 hd-Cons-tl
length-append less-le list.size(3) nth-Cons-Suc nth-append-length-plus takeWhile-dropWhile-id)

 have i + 1 < length (filter P ?xs)
  using Suc(2)
  by auto
 then obtain k1 k2
  where k1 < k2 k2 < length ?xs
    filter P ?xs ! i = ?xs ! k1
    filter P ?xs ! (i + 1) = ?xs ! k2
    P (?xs ! k1)
    P (?xs ! k2)
    ∀ k'. k1 < k' ∧ k' < k2 ⟶ ¬ P (?xs ! k')
  using Suc(1)[of ?xs]
  by auto
 show ?case
 proof (rule-tac x=k1+length ?t+1 in exI, rule-tac x=k2+length ?t+1 in exI,
```

safe)
show *k1 + length ?t + 1 < k2 + length ?t + 1*
using ‹*k1 < k2*›
by *simp*
next
have *k2 + length ?t + 1 < length ?xs + 1 + length ?t*
using ‹*k2 < length ?xs*›
by *simp*
then show *k2 + length ?t + 1 < length xs*
using ‹*?xs ≠ []*›
by (*metis One-nat-def Suc-eq-plus1 Suc-pred add.commute add-lessD1 length-append length-greater-0-conv length-tl less-diff-conv takeWhile-dropWhile-id*)
next
show *P (xs ! (k1 + length ?t + 1))*
using ‹*P (?xs ! k1)*› ‹*k1 < k2*› ‹*k2 < length ?xs*› ∗
by (*metis Suc-eq-plus1 add.commute add-Suc-right hd-Cons-tl length-greater-0-conv length-tl list.size(3) not-less-zero nth-Cons-Suc nth-append-length-plus takeWhile-dropWhile-id zero-less-diff*)
next
show *P (xs ! (k2 + length (takeWhile (λx. ¬ P x) xs) + 1))*
using ‹*P (?xs ! k2)*› ‹*k2 < length ?xs*› ∗
by (*metis Suc-eq-plus1 add.commute add-Suc-right hd-Cons-tl length-greater-0-conv length-tl list.size(3) not-less-zero nth-Cons-Suc nth-append-length-plus takeWhile-dropWhile-id zero-less-diff*)
next
fix *k′*
assume *k1 + length ?t + 1 < k′ k′ < k2 + length ?t + 1 P (xs ! k′)*
then have *k1 < k′ − (length ?t + 1) k′ − (length ?t + 1) < k2*
using ‹*k1 < k2*› ‹*k2 < length ?xs*›
by *linarith*+
moreover
have *length ?t + (k′ − (length ?t + 1)) + 1 < length xs*
using ‹*k2 < length (tl (dropWhile (λx. ¬ P x) xs))*›
by (*smt ab-semigroup-add-class.add-ac(1) add.commute add-lessD1 add-less-cancel-left calculation(2) length-append length-tl less-diff-conv less-trans-Suc plus-1-eq-Suc takeWhile-dropWhile-id*)
then have *P (?xs ! (k′ − (length ?t + 1)))*
using ∗[*rule-format*, *of k′ − (length ?t + 1)*] ‹*P (xs ! k′)*›
by (*metis Suc-eq-plus1 add-Suc add-diff-inverse-nat calculation(1) nat-diff-split not-less-zero*)
ultimately

show *False*
using ‹∀ *k'*. *k1* < *k'* ∧ *k'* < *k2* ⟶ ¬ *P* (*?xs* ! *k'*)›[*rule-format*, *of k'* − (*length ?t* + *1*)] ‹*k1* < *k2*› ‹*k2* < *length ?xs*›
by *simp*
next
show *filter P xs* ! (*Suc i*) = *xs* ! (*k1* + *length ?t* + *1*)
proof−
have *filter P xs* ! (*Suc i*) = *filter P ?d* ! (*Suc i*)
by *simp*
also have ... = *filter P* (*tl ?d*) ! *i*
using ‹*?xs* ≠ []› ‹*i* + *1* < *length* (*filter P ?xs*)›
by (*metis add-lessD1 filter-tl hd-dropWhile list.sel*(*2*) *nth-tl*)
finally
show *?thesis*
using ‹*filter P ?xs* ! *i* = *?xs* ! *k1*› ∗
using ‹*k1* < *k2*› ‹*k2* < *length ?xs*›
by (*smt Suc-eq-plus1 add.commute add-Suc-right add-lessD1 add-less-cancel-left length-append length-tl less-diff-conv less-trans-Suc takeWhile-dropWhile-id*)
qed
next
show *filter P xs* ! (*Suc i* + *1*) = *xs* ! (*k2* + *length ?t* + *1*)
proof−
have *filter P xs* ! (*Suc i* + *1*) = *filter P ?d* ! (*Suc i* + *1*)
by *simp*
also have ... = *filter P* (*tl ?d*) ! (*Suc i*)
using ‹*?xs* ≠ []› ‹*i* + *1* < *length* (*filter P ?xs*)›
by (*metis add.commute filter-tl hd-dropWhile nth-tl plus-1-eq-Suc tl-Nil*)
finally
show *?thesis*
using ‹*filter P ?xs* ! (*i* + *1*) = *?xs* ! *k2*› ∗
using ‹*k1* < *k2*› ‹*k2* < *length ?xs*›
by (*smt Suc-eq-plus1 add.commute add-Suc-right add-lessD1 add-less-cancel-left length-append length-tl less-diff-conv less-trans-Suc takeWhile-dropWhile-id*)
qed
qed
qed

Unlabeled states

type-synonym *state* = *nat list*

definition *initial-state* :: *nat* ⇒ *state* **where**
initial-state n = (*replicate* (*n* + *1*) *0*) [*0* := *n*]

definition *final-state* :: *nat* ⇒ *state* **where**
final-state n = (*replicate* (*n* + *1*) *0*) [*n* := *n*]

definition *valid-state* :: *nat* ⇒ *state* ⇒ *bool* **where**
valid-state n state ⟷ *length state* = *n* + *1* ∧ *sum-list state* = *n*

definition *move* :: *nat* ⇒ *nat* ⇒ *state* ⇒ *state* **where**
move p1 p2 state =
(*let k1* = *state* ! *p1*;
k2 = *state* ! *p2*
in state [*p1* := *k1* − *1*, *p2* := *k2* + *1*])

definition *valid-move′* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *state* ⇒ *state* ⇒ *bool* **where**
valid-move′ n p1 p2 state state′ ⟷
(*let k1* = *state* ! *p1*
in k1 > *0* ∧ *p1* < *p2* ∧ *p2* ≤ *p1* + *k1* ∧ *p2* ≤ *n* ∧
state′ = *move p1 p2 state*)

definition *valid-move* :: *nat* ⇒ *state* ⇒ *state* ⇒ *bool* **where**
valid-move n state state′ ⟷
(∃ *p1 p2*. *valid-move′ n p1 p2 state state′*)

definition *valid-moves* **where**
valid-moves n states ⟷
(∀ *i* < *length states* − *1*. *valid-move n* (*states* ! *i*) (*states* ! (*i* + *1*)))

definition *valid-game* **where**
valid-game n states ⟷
length states ≥ *2* ∧
hd states = *initial-state n* ∧
last states = *final-state n* ∧
valid-moves n states

lemma *valid-state-initial-state* [*simp*]:
shows *valid-state n* (*initial-state n*)

```
  by (simp add: initial-state-def valid-state-def)

lemma valid-move-valid-state:
  assumes valid-state n state valid-move n state state'
  shows valid-state n state'
proof−
  obtain p1 p2
    where ∗: 0 < state ! p1 p1 < p2 p2 ≤ p1 + state ! p1 p2 ≤ n state' = state[p1
:= state ! p1 − 1, p2 := state ! p2 + 1]
    using assms
    unfolding valid-move-def valid-move'-def move-def Let-def
    by auto
  then have sum-list state > 0
    using assms(1) valid-state-def
    by auto
  then have sum-list (state[p1 := state ! p1 − 1, p2 := state ! p2 + 1]) =
sum-list state
    using ∗ assms
    using sum-list-update[of p1 state state ! p1 − 1]
    using sum-list-update[of p2 state[p1 := state ! p1 − 1] state ! p2 + 1]
    unfolding valid-state-def
    by auto
  then show ?thesis
    using ⟨valid-state n state⟩ ∗
    by (simp add: valid-state-def)
qed

lemma valid-moves-Nil [simp]:
  shows valid-moves n []
  by (simp add: valid-moves-def)

lemma valid-moves-Single [simp]:
  shows valid-moves n [state]
  by (simp add: valid-moves-def)

lemma valid-moves-Cons [simp]:
  shows valid-moves n (state1 # state2 # states) ⟷
        valid-move n state1 state2 ∧ valid-moves n (state2 # states)
  unfolding valid-moves-def
  by (auto simp add: nth-Cons split: nat.split)
```

lemma *valid-moves-valid-states*:
assumes *valid-moves n states valid-state n* (*hd states*)
shows ∀ *state* ∈ *set states*. *valid-state n state*
using *assms*
proof (*induction states*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a states*)
then show *?case*
by (*metis list.sel*(*1*) *list.set-cases set-ConsD valid-moves-Cons valid-move-valid-state*)
qed

lemma *valid-game-valid-states*:
assumes *valid-game n states*
shows ∀ *state* ∈ *set states*. *valid-state n state*
using *assms*
unfolding *valid-game-def*
using *valid-moves-valid-states*
by *fastforce*

definition *move-positions* **where**
move-positions state state′ =
(*THE* (*p1*, *p2*). *valid-move′* (*length state* − *1*) *p1 p2 state state′*)

lemma *move-positions-unique*:
assumes *valid-state n state valid-move n state state′*
shows ∃! (*p1*, *p2*). *valid-move′ n p1 p2 state state′*
proof−
have *length state* = *n* + *1*
using *assms*
unfolding *valid-state-def*
by *simp*

have ∃! *p1*. *p1* < *length state* ∧ *state* ! *p1* > *0* ∧ *state′* ! *p1* = *state* ! *p1* − *1*
using *assms*
unfolding *valid-state-def valid-move-def valid-move′-def Let-def move-def*
by (*smt add.right-neutral add-Suc-right add-diff-cancel-left′ le-SucI less-imp-Suc-add*

less-le-trans list-update-swap n-not-Suc-n nat.simps(3) nth-list-update-eq nth-list-update-neq plus-1-eq-Suc)
then have ∗: ∃! *p1*. *p1* ≤ *n* ∧ *state* ! *p1* > *0* ∧ *state'* ! *p1* = *state* ! *p1* − *1*
using ‹*length state* = *n* + *1*›
by (*metis Nat.le-diff-conv2 Suc-leI add.commute add-diff-cancel-right' le-add2 le-imp-less-Suc plus-1-eq-Suc*)

have ∃! *p2*. *p2* < *length state* ∧ *state'* ! *p2* = *state* ! *p2* + *1*
using *assms*
unfolding *valid-state-def valid-move-def valid-move'-def Let-def move-def*
by (*metis Groups.add-ac(2) diff-le-self le-imp-less-Suc length-list-update n-not-Suc-n nat-neq-iff nth-list-update-eq nth-list-update-neq plus-1-eq-Suc*)
then have ∗∗: ∃! *p2*. *p2* ≤ *n* ∧ *state'* ! *p2* = *state* ! *p2* + *1*
using ‹*length state* = *n* + *1*›
by (*simp add*: *discrete*)

obtain *p1 p2* **where** *valid-move' n p1 p2 state state'*
using *assms*
unfolding *valid-move-def*
by *auto*
show *?thesis*
proof
show *case* (*p1*, *p2*) *of* (*p1*, *p2*) ⇒ *valid-move' n p1 p2 state state'*
using ‹*valid-move' n p1 p2 state state'*›
by *simp*
next
fix *x*
assume *case x of* (*p1'*, *p2'*) ⇒ *valid-move' n p1' p2' state state'*
then obtain *p1' p2'* **where** *x* = (*p1'*, *p2'*) *valid-move' n p1' p2' state state'*
by *auto*
then show *x* = (*p1*, *p2*)
using ‹*valid-move' n p1 p2 state state'*› ∗ ∗∗ ‹*length state* = *n* + *1*›
unfolding *valid-move'-def move-def Let-def*
by (*metis Nat.add-0-right One-nat-def add-Suc-right le-imp-less-Suc le-less-trans length-list-update less-imp-le-nat nat-neq-iff nth-list-update-eq nth-list-update-neq*)
qed
qed

lemma *valid-move'-move-positions*:
assumes *valid-state n state valid-move' n p1 p2 state state'*

```
  shows (p1, p2) = move-positions state state'
proof−
  have ∗: (THE x. let (p1', p2') = x in valid-move' (length state − 1) p1' p2'
state state') = (p1, p2)
  proof (rule the-equality)
    show let (p1', p2') = (p1, p2) in valid-move' (length state − 1) p1' p2' state
state'
      using assms
      unfolding valid-state-def valid-move-def Let-def
      by auto
  next
    fix x
   assume let (p1', p2') = x in valid-move' (length state − 1) p1' p2' state state'
    then show x = (p1, p2)
      using move-positions-unique[of n state state'] assms
      unfolding valid-state-def valid-move-def
      by auto
  qed
  then show ?thesis
    unfolding move-positions-def Let-def
    by auto
qed

lemma move-positions-valid-move':
  assumes valid-state n state valid-move n state state'
          (p1, p2) = move-positions state state'
  shows valid-move' n p1 p2 state state'
  using assms
  by (metis fstI sndI valid-move-def valid-move'-move-positions)
```

Labeled states

```
type-synonym labeled-state = (nat set) list

definition initial-labeled-state :: nat ⇒ labeled-state where
  initial-labeled-state n  = (replicate (n+1) {}) [0 := {0..<n}]

definition final-labeled-state :: nat ⇒ labeled-state where
  final-labeled-state n  = (replicate (n+1) {}) [n := {0..<n}]
```

definition *valid-labeled-state* :: *nat* ⇒ *labeled-state* ⇒ *bool* **where**
valid-labeled-state n l-state ⟷
length l-state = *n*+*1* ∧
(∀ *i j*. *i* < *j* ∧ *j* ≤ *n* ⟶ *l-state* ! *i* ∩ *l-state* ! *j* = {}) ∧
(⋃ (*set l-state*)) = {*0*..<*n*}

definition *labeled-move* **where**
labeled-move p1 p2 stone l-state =
(*let ss1* = *l-state* ! *p1*;
ss2 = *l-state* ! *p2*
in l-state [*p1* := *ss1* − {*stone*}, *p2* := *ss2* ∪ {*stone*}])

definition *valid-labeled-move′* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *labeled-state* ⇒ *labeled-state* ⇒ *bool* **where**
valid-labeled-move′ n p1 p2 stone l-state l-state′ ⟷
(*let ss1* = *l-state* ! *p1*
in p1 < *p2* ∧ *p2* ≤ *p1* + *card ss1* ∧ *p2* ≤ *n* ∧
stone ∈ *ss1* ∧ *l-state′* = *labeled-move p1 p2 stone l-state*)

definition *valid-labeled-move* :: *nat* ⇒ *labeled-state* ⇒ *labeled-state* ⇒ *bool* **where**

valid-labeled-move n l-state l-state′ ⟷
(∃ *p1 p2 stone*. *valid-labeled-move′ n p1 p2 stone l-state l-state′*)

definition *valid-labeled-moves* **where**
valid-labeled-moves n l-states ⟷
(∀ *i* < *length l-states* − *1*. *valid-labeled-move n* (*l-states* ! *i*) (*l-states* ! (*i* + *1*)))

definition *valid-labeled-game* **where**
valid-labeled-game n l-states ⟷
length l-states ≥ *2* ∧
hd l-states = *initial-labeled-state n* ∧
last l-states = *final-labeled-state n* ∧
valid-labeled-moves n l-states

lemma *valid-labeled-state-initial-labeled-state* [*simp*]:
shows *valid-labeled-state n* (*initial-labeled-state n*)
unfolding *valid-labeled-state-def initial-labeled-state-def*
by *auto*

lemma *valid-labeled-state-final-labeled-state* [*simp*]:
shows *valid-labeled-state n* (*final-labeled-state n*)
proof−
have (*replicate* (*Suc n*) {})[*n* := {*0*..<*n*}] = (*replicate n* {}) @ [{*0*..<*n*}]
by (*metis length-replicate list-update-length replicate-Suc replicate-append-same*)
then show *?thesis*
unfolding *valid-labeled-state-def final-labeled-state-def*
by (*auto simp del*: *replicate-Suc simp add*: *nth-append*)
qed

lemma *valid-labeled-move-valid-labeled-state*:
assumes *valid-labeled-state n l-state valid-labeled-move n l-state l-state′*
shows *valid-labeled-state n l-state′*
proof−
from *assms* **obtain** *p1 p2 stone* **where**
∗∗: *p1* < *p2 p2* ≤ *p1* + *card* (*l-state* ! *p1*) *p2* ≤ *n length l-state* = *n*+*1* ⋃ (*set l-state*) = {*0*..<*n*} ∀ *i j*. *i* < *j* ∧ *j* ≤ *n* ⟶ *l-state* ! *i* ∩ *l-state* ! *j* = {}
stone ∈ *l-state* ! *p1 l-state′* = *l-state*[*p1* := *l-state* ! *p1* − {*stone*}, *p2* := *l-state* ! *p2* ∪ {*stone*}]
unfolding *valid-labeled-move-def valid-labeled-move′-def valid-labeled-state-def Let-def labeled-move-def*
by *auto*

then have ∗: ∀ *i* ≤ *n*. *l-state′* ! *i* = (*if i* = *p1 then l-state* ! *p1* − {*stone*}
else if i = *p2 then l-state* ! *p2* ∪ {*stone*}
else l-state ! *i*) *length l-state′* = *n* + *1*
by *auto*

have *stone* ∉ *l-state* ! *p2*
using ⟨∀ *i j*. *i* < *j* ∧ *j* ≤ *n* ⟶ *l-state* ! *i* ∩ *l-state* ! *j* = {}⟩ ⟨*stone* ∈ *l-state* ! *p1*⟩
using ⟨*p1* < *p2*⟩ ⟨*p2* ≤ *n*⟩
by (*metis Collect-mem-eq IntI empty-Collect-eq*)

have ∀ *i* ≤ *n*. *i* ≠ *p1* ⟶ *stone* ∉ *l-state* ! *i*
using ⟨∀ *i j*. *i* < *j* ∧ *j* ≤ *n* ⟶ *l-state* ! *i* ∩ *l-state* ! *j* = {}⟩ ⟨*stone* ∈ *l-state* ! *p1*⟩
using ⟨*p1* < *p2*⟩ ⟨*p2* ≤ *n*⟩
by (*metis disjoint-iff-not-equal le-less-trans less-imp-le-nat nat-neq-iff*)

```
  have ⋃ (set l-state') = ⋃ (set l-state)
  proof safe
    fix x X
    assume x ∈ X X ∈ set l-state'
    then obtain i where x ∈ l-state' ! i i ≤ n
      using ⟨length l-state' = n+1⟩
     by (metis One-nat-def add.right-neutral add-Suc-right in-set-conv-nth le-simps(2))

    then show x ∈ ⋃ (set l-state)
      using * ⟨stone ∈ l-state ! p1⟩ **
       by (smt Diff-iff One-nat-def Un-insert-right add.right-neutral add-Suc-right
boolean-algebra-cancel.sup0 insertE le-imp-less-Suc le-less-trans less-imp-le-nat mem-simps(9)
nth-mem)
  next
    fix x X
    assume x ∈ X X ∈ set l-state
    then obtain i where i ≤ n x ∈ l-state ! i
      using ⟨length l-state = n + 1⟩
      by (metis add.commute in-set-conv-nth le-simps(2) plus-1-eq-Suc)
    show x ∈ ⋃ (set l-state')
    proof (cases i = p1)
      case True
      then have x ∈ l-state' ! p1 ∨ x ∈ l-state' ! p2
        using * ⟨p1 < p2⟩ ⟨p2 ≤ n⟩ ⟨x ∈ l-state ! i⟩
        by auto
      then show ?thesis
        using ⟨p1 < p2⟩ ⟨p2 ≤ n⟩
        using *(2) mem-simps(9) nth-mem
        by auto
    next
      case False
      then have x ∈ l-state' ! i
        using * ⟨p1 < p2⟩ ⟨p2 ≤ n⟩ ⟨x ∈ l-state ! i⟩
        using ⟨i ≤ n⟩ by auto
      then show ?thesis
       by (metis *(2) One-nat-def Sup-upper ⟨i ≤ n⟩ add.right-neutral add-Suc-right
le-imp-less-Suc nth-mem subsetD)
    qed
  qed
```

```
  moreover

  have ∀ i j. i < j ∧ j ≤ n ⟶ l-state' ! i ∩ l-state' ! j = {}
  proof safe
    fix i j x
    assume ***: i < j j ≤ n x ∈ l-state' ! i x ∈ l-state' ! j
    then have False
      using * ‹∀ i j. i < j ∧ j ≤ n ⟶ l-state ! i ∩ l-state ! j = {}›
      using ‹stone ∈ l-state ! p1› ‹stone ∉ l-state ! p2› ‹∀ i ≤ n. i ≠ p1 ⟶ stone
∉ l-state ! i›
      using ‹length l-state = n+1› ‹length l-state' = n+1› ‹p1 < p2› ‹p2 ≤ n›
      apply (cases j = p2)
     apply (smt Diff-insert-absorb Diff-subset IntI Un-insert-right boolean-algebra-cancel.sup0
empty-iff insertE less-imp-le-nat less-le-trans mk-disjoint-insert nat-neq-iff subsetD)
      apply (smt Un-insert-right boolean-algebra-cancel.sup0 disjoint-iff-not-equal
insert-Diff insert-iff less-imp-le-nat less-le-trans)
      done
    then show x ∈ {}
      by simp
  qed

  ultimately

  show ?thesis
    unfolding valid-labeled-state-def
    using assms
   unfolding valid-labeled-move-def Let-def valid-labeled-move'-def labeled-move-def
valid-labeled-state-def
    by auto
qed

lemma valid-labeled-moves-valid-labeled-states:
  assumes valid-labeled-moves n l-states valid-labeled-state n (hd l-states)
  shows ∀ state ∈ set l-states. valid-labeled-state n state
  using assms
proof (induction l-states)
  case Nil
  then show ?case
    by simp
```

next
case (*Cons a states*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *Groups.add-ac*(*2*) *hd-Cons-tl length-greater-0-conv length-tl less-diff-conv list.inject list.set-cases list.simps*(*3*) *nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc valid-labeled-moves-def valid-labeled-move-valid-labeled-state*)
qed

lemma *valid-labeled-game-valid-labeled-states*:
assumes *valid-labeled-game n states*
shows ∀ *state* ∈ *set states*. *valid-labeled-state n state*
using *assms*
unfolding *valid-labeled-game-def*
using *valid-labeled-moves-valid-labeled-states*
by *fastforce*

definition *labeled-move-positions* **where**
labeled-move-positions state state′ =
(*THE* (*p1*, *p2*, *stone*). *valid-labeled-move′* (*length state* − *1*) *p1 p2 stone state state′*)

lemma *labeled-move-positions-unique*:
assumes *valid-labeled-state n state valid-labeled-move n state state′*
shows ∃! (*p1*, *p2*, *stone*). *valid-labeled-move′ n p1 p2 stone state state′*
proof−
obtain *p1 p2 stone* **where** ∗: *valid-labeled-move′ n p1 p2 stone state state′*
using *assms*
unfolding *valid-labeled-move-def*
by *auto*
show *?thesis*
proof
show *case* (*p1*, *p2*, *stone*) *of* (*p1*, *p2*, *stone*) ⇒ *valid-labeled-move′ n p1 p2 stone state state′*
using ∗
by *auto*
next
fix *x* :: *nat* × *nat* × *nat*
obtain *p1′ p2′ stone′* **where** *x*: *x* = (*p1′*, *p2′*, *stone′*)
by (*cases x*)
assume *case x of* (*p1*, *p2*, *stone*) ⇒ *valid-labeled-move′ n p1 p2 stone state*

state′
then have ∗∗: *valid-labeled-move′ n p1′ p2′ stone′ state state′*
using *x*
by *simp*
have ∗: *p1 < p2 p2 ≤ n stone < n stone ∈ state ! p1 stone ∉ state′ ! p1 stone ∉ state ! p2 stone ∈ state′ ! p2*
∀ *stone″ p. p ≤ n ∧ stone″ < n ∧ stone″ ≠ stone ⟶ (stone″ ∈ state ! p ⟷ stone″ ∈ state′ ! p)*
using ∗ *assms*(*1*)
unfolding *valid-labeled-state-def valid-labeled-move′-def Let-def labeled-move-def*
by (*auto simp add*: *nth-list-update*)

have ∗∗: *p1′ < p2′ p2′ ≤ n stone′ < n stone′ ∈ state ! p1′ stone′ ∉ state′ ! p1′ stone′ ∉ state ! p2′ stone′ ∈ state′ ! p2′*
∀ *stone″ p. p ≤ n ∧ stone″ < n ∧ stone″ ≠ stone′ ⟶ (stone″ ∈ state ! p ⟷ stone″ ∈ state′ ! p)*
using ∗∗ *assms*(*1*)
unfolding *valid-labeled-state-def valid-labeled-move′-def Let-def labeled-move-def*
by (*auto simp add*: *nth-list-update*)

have *stone = stone′*
using ∗ ∗∗
by *auto*

have *disj*: ∀ *i j. i < j ∧ j ≤ n ⟶ state ! i ∩ state ! j = {}*
using *assms*(*1*)
unfolding *valid-labeled-state-def*
by *auto*

have *p1 = p1′*
using ∗(*4*) ∗∗(*4*) ‹*stone = stone′*› ∗(*1−2*) ∗∗(*1−2*)
using *disj*[*rule-format*, *of p1 p1′*]
using *disj*[*rule-format*, *of p1′ p1*]
by *force*

have *valid-labeled-state n state′*
using *assms*(*1*) *assms*(*2*) *valid-labeled-move-valid-labeled-state* **by** *blast*
then have *disj′*: ∀ *i j. i < j ∧ j ≤ n ⟶ state′ ! i ∩ state′ ! j = {}*
unfolding *valid-labeled-state-def*
by *auto*

```
    have p2 = p2'
      using *(7) **(7) ‹stone = stone'› *(2) **(2)
      using disj'[rule-format, of p2 p2']
      using disj'[rule-format, of p2' p2]
      by force

    then show x = (p1, p2, stone)
      using x ‹stone = stone'› ‹p1 = p1'› ‹p2 = p2'›
      by auto
  qed
qed

lemma labeled-move-positions:
  assumes valid-labeled-state n state valid-labeled-move' n p1 p2 stone state state'
  shows labeled-move-positions state state' = (p1, p2, stone)
  using assms
  using labeled-move-positions-unique[OF assms(1), of state']
 unfolding labeled-move-positions-def valid-labeled-state-def valid-labeled-move-def
  by auto (smt case-prodI the-equality)

lemma labeled-move-positions-valid-move':
  assumes valid-labeled-state n state valid-labeled-move n state state'
          labeled-move-positions state state' = (p1, p2, stone)
  shows valid-labeled-move' n p1 p2 stone state state'
 using assms(1) assms(2) assms(3) labeled-move-positions valid-labeled-move-def
  by auto

definition stone-position :: labeled-state ⇒ nat ⇒ nat where
  stone-position l-state stone =
    (THE k. k < length l-state ∧ stone ∈ l-state ! k)

lemma stone-position-unique:
  assumes valid-labeled-state n l-state stone < n
  shows ∃! k. k < length l-state ∧ stone ∈ l-state ! k
proof−
  from assms have stone ∈ ⋃ (set l-state)
    unfolding valid-labeled-state-def
    by auto
  then obtain k where *: k < length l-state stone ∈ l-state ! k
```

```
    by (metis UnionE in-set-conv-nth)
  then have ∀ k'. k' < length l-state ∧ stone ∈ l-state ! k' ⟶ k = k'
    using assms
    unfolding valid-labeled-state-def
    by (metis IntI Suc-eq-plus1 empty-iff le-simps(2) nat-neq-iff)
  then show ?thesis
    using ∗
    by auto
qed

lemma stone-position:
  assumes valid-labeled-state n l-state stone < n
  shows stone-position l-state stone ≤ n ∧
        stone ∈ l-state ! (stone-position l-state stone)
  using assms stone-position-unique[OF assms]
  using theI[of λ k. k < length l-state ∧ stone ∈ l-state ! k]
  unfolding valid-labeled-state-def stone-position-def
 by (metis (mono-tags, lifting) One-nat-def add.right-neutral add-Suc-right le-simps(2))

lemma stone-positionI:
  assumes valid-labeled-state n l-state stone < n
          k < length l-state stone ∈ l-state ! k
  shows stone-position l-state stone = k
  unfolding stone-position-def
  using assms stone-position-unique
  by blast

lemma valid-labeled-move'-stone-positions:
   assumes valid-labeled-state n l-state valid-labeled-move' n p1 p2 stone l-state
l-state'
  shows stone-position l-state stone = p1 ∧ stone-position l-state' stone = p2
proof safe
  show stone-position l-state stone = p1
  proof (rule stone-positionI)
    show valid-labeled-state n l-state stone < n p1 < length l-state stone ∈ l-state
! p1
      using assms
      unfolding valid-labeled-state-def valid-labeled-move'-def Let-def
      by auto
  qed
```

next
show *stone-position l-state′ stone = p2*
proof (*rule stone-positionI*)
show *valid-labeled-state n l-state′*
using *assms(1) assms(2) valid-labeled-move-def valid-labeled-move-valid-labeled-state*
by *blast*
next
show *stone < n p2 < length l-state′ stone ∈ l-state′ ! p2*
using *assms*
unfolding *valid-labeled-state-def valid-labeled-move′-def Let-def labeled-move-def*
by *auto*
qed
qed

lemma *valid-labeled-move′-stone-positions-other*:
assumes *valid-labeled-state n l-state valid-labeled-move′ n p1 p2 stone l-state l-state′*
shows ∀ *stone′. stone′ ≠ stone ∧ stone′ < n ⟶ stone-position l-state′ stone′ = stone-position l-state stone′*
proof *safe*
fix *stone′*
assume *stone′ < n stone′ ≠ stone*
show *stone-position l-state′ stone′ = stone-position l-state stone′*
proof (*rule stone-positionI*)
show *stone′ < n*
by *fact*
next
show *valid-labeled-state n l-state′*
using *assms*
using *valid-labeled-move-def valid-labeled-move-valid-labeled-state*
by *blast*
next
show *stone-position l-state stone′ < length l-state′*
using ‹*stone′ < n*› *assms(1−2) stone-position*[*of n l-state stone′*]
unfolding *valid-labeled-state-def*
by (*metis Suc-eq-plus1 labeled-move-def le-imp-less-Suc length-list-update valid-labeled-move′-def*)
next
show *stone′ ∈ l-state′ ! stone-position l-state stone′*
proof−

```
    have stone' ∈ l-state ! stone-position l-state stone'
        stone-position l-state stone' < length l-state
      using ⟨stone' < n⟩ assms(1−2) stone-position[of n l-state stone']
      unfolding valid-labeled-state-def
      by auto
    then show ?thesis
      using ⟨stone' ≠ stone⟩ ⟨valid-labeled-move' n p1 p2 stone l-state l-state'⟩
      unfolding valid-labeled-move'-def labeled-move-def Let-def
        by (metis (no-types, lifting) Un-insert-right boolean-algebra-cancel.sup0
insert-Diff insert-iff length-list-update nth-list-update-eq nth-list-update-neq)
  qed
 qed
qed
```

Unlabel

```
definition unlabel :: labeled-state ⇒ state where
 unlabel = map card

lemma unlabel-initial [simp]:
 shows unlabel (initial-labeled-state n) = initial-state n
 unfolding initial-labeled-state-def initial-state-def unlabel-def
 by auto

lemma unlabel-final [simp]:
 shows unlabel (final-labeled-state n) = final-state n
 unfolding final-labeled-state-def final-state-def unlabel-def
 by (metis card-atLeastLessThan card-empty diff-zero map-replicate map-update)

lemma unlabel-valid:
 assumes valid-labeled-state n l-state
 shows valid-state n (unlabel l-state)
 unfolding valid-state-def unlabel-def
proof
 let ?state = map card l-state
 show length ?state = n + 1
  using assms
  by (simp add: valid-labeled-state-def)

 show sum-list ?state = n
```

```
proof−
  let ?s = filter (λ y. card y ≠ 0) l-state

  have (∑ x ← l-state. card x) = (∑ x ← ?s. card x)
    by (metis (mono-tags, lifting) sum-list-map-filter)
  also have ... = (∑ x ∈ set ?s. card x)
  proof−
    have ∀ i j. i < j ∧ j < length l-state ⟶ l-state ! i ∩ l-state ! j = {}
      using assms
      unfolding valid-labeled-state-def
      by simp
    then have distinct ?s
    proof (induction l-state)
      case Nil
      then show ?case
        by simp
    next
      case (Cons a l-state)
      have ∀ i j. i < j ∧ j < length l-state ⟶ l-state ! i ∩ l-state ! j = {}
        using Cons(2)
        by (metis One-nat-def Suc-eq-plus1 Suc-less-eq list.size(4) nth-Cons-Suc)
      then have distinct (filter (λy. card y ≠ 0) l-state)
        using Cons(1)
        by simp
      moreover
      have card a > 0 ⟶ a ∉ set l-state
      proof safe
        assume card a > 0 a ∈ set l-state
        show False
          using Cons(2)[rule-format, of 0] ‹0 < card a› ‹a ∈ set l-state›
          by (metis card-empty in-set-conv-nth inf.idem le-simps(2) length-Cons
not-le nth-Cons-0 nth-Cons-Suc zero-less-Suc)
      qed
      ultimately
      show ?case
        using Cons
        by auto
    qed
    then show ?thesis
      using sum-list-distinct-conv-sum-set by blast
```

```
qed
also have ... = card (⋃ (set ?s))
proof−
  have ∀ i∈set ?s. finite (id i)
    using assms
    unfolding valid-labeled-state-def
    by fastforce
  moreover
  have ∀ i∈set ?s.
        ∀ j∈set ?s. i ≠ j ⟶ id i ∩ id j = {}
  proof (rule ballI, rule ballI, rule impI)
    fix i j
    assume i ∈ set ?s j ∈ set ?s i ≠ j
    then obtain i' j' where i = l-state ! i' j = l-state ! j' i' ≤ n j' ≤ n
      using assms
      unfolding valid-labeled-state-def
     by (metis Suc-eq-plus1 filter-is-subset in-set-conv-nth le-simps(2) subsetD)
    then show id i ∩ id j = {}
      using assms ‹i ≠ j›
      unfolding valid-labeled-state-def
      by (metis disjoint-iff-not-equal id-apply nat-neq-iff)
  qed
  ultimately
  show ?thesis
    using card-UN-disjoint[of set ?s id, symmetric]
    by simp
qed
also have card (⋃ (set ?s)) = card (⋃ (set l-state))
proof−
  have ⋃ (set ?s) = ⋃ (set l-state)
  proof
    show ⋃ (set l-state) ⊆ ⋃ (set ?s)
    proof safe
      fix x X
      assume ∗: x ∈ X X ∈ set l-state
      then have card X ≠ 0
        using assms
        unfolding valid-labeled-state-def
        using Union-upper finite-subset
        by fastforce
```

```
        then show x ∈ ⋃ (set ?s)
          using ∗
          by auto
      qed
    qed auto
    then show ?thesis
      by simp
  qed
  finally
  show ?thesis
    using assms
    unfolding valid-labeled-state-def
    by simp
 qed
qed

lemma unlabel-valid-move′:
  assumes valid-labeled-state n l-state valid-labeled-move′ n p1 p2 stone l-state
l-state′
 shows valid-move′ n p1 p2 (unlabel l-state) (unlabel l-state′) ∧
       unlabel l-state′ = move p1 p2 (unlabel l-state)
proof−
 from assms have
   p1 < p2 p2 ≤ p1 + card (l-state ! p1) p2 ≤ n length l-state = n+1 ⋃ (set
l-state) = {0..<n} ∀ i j. i < j ∧ j ≤ n ⟶ l-state ! i ∩ l-state ! j = {}
   stone ∈ l-state ! p1 l-state′ = l-state[p1 := l-state ! p1 − {stone}, p2 := l-state
! p2 ∪ {stone}]
   unfolding valid-labeled-move-def valid-labeled-move′-def valid-labeled-state-def
unlabel-def Let-def labeled-move-def
   by auto

 have finite (l-state ! p1) ∧ finite (l-state ! p2)
   using ⟨⋃ (set l-state) = {0..<n}⟩
   using ⟨length l-state = n + 1⟩ ⟨p1 < p2⟩ ⟨p2 ≤ n⟩
  by (metis One-nat-def Union-upper add.right-neutral add-Suc-right finite-atLeastLessThan
finite-subset le-imp-less-Suc le-less-trans less-imp-le-nat nth-mem)

 have stone ∉ l-state ! p2
   using ⟨stone ∈ l-state ! p1⟩ ⟨∀ i j. i < j ∧ j ≤ n ⟶ l-state ! i ∩ l-state ! j =
{}⟩
```

using ‹*length l-state* = *n* + *1*› ‹*p1* < *p2*› ‹*p2* ≤ *n*›
by (*metis Collect-empty-eq Collect-mem-eq IntI*)

have *card* (*l-state* ! *p1*) > *0 length l-state'* = *length l-state*
card (*l-state'* ! *p1*) = *card* (*l-state* ! *p1*) − *1* *card* (*l-state'* ! *p2*) = *card* (*l-state* ! *p2*) + *1*
∀ *p*. *p* ≤ *n* ∧ *p* ≠ *p1* ∧ *p* ≠ *p2* ⟶ *card* (*l-state'* ! *p*) = *card* (*l-state* ! *p*)
using ‹*finite* (*l-state* ! *p1*) ∧ *finite* (*l-state* ! *p2*)› ‹*stone* ∈ *l-state* ! *p1*›
using ‹*stone* ∉ *l-state* ! *p2*› ‹*l-state'* = *l-state*[*p1* := *l-state* ! *p1* − {*stone*}, *p2* := *l-state* ! *p2* ∪ {*stone*}]›
using ‹*length l-state* = *n* + *1*› ‹*p1* < *p2*› ‹*p2* ≤ *n*›
using *card-0-eq*
by − (*blast*, *simp*+)

then show *?thesis*
using ‹*length l-state* = *n* + *1*› ‹*p1* < *p2*› ‹*p2* ≤ *p1* + *card* (*l-state* ! *p1*)› ‹*p2* ≤ *n*›
using ‹*l-state'* = *l-state*[*p1* := *l-state* ! *p1* − {*stone*}, *p2* := *l-state* ! *p2* ∪ {*stone*}]›
unfolding *unlabel-def valid-move'-def*
by (*auto simp add*: *move-def map-update*)
qed

lemma *unlabel-valid-move*:
assumes *valid-labeled-state n l-state valid-labeled-move n l-state l-state'*
shows *valid-move n* (*unlabel l-state*) (*unlabel l-state'*)
using *assms*(*2*) *unlabel-valid-move'*[*OF assms*(*1*)]
unfolding *valid-labeled-move-def valid-move-def Let-def*
by *force*

Labeled move max stone

definition *valid-labeled-move-max-stone* :: *nat* ⇒ *labeled-state* ⇒ *labeled-state* ⇒ *bool* **where**
valid-labeled-move-max-stone n l-state l-state' ⟷
(∃ *p1 p2*. *valid-labeled-move' n p1 p2* (*Max* (*l-state* ! *p1*)) *l-state l-state'*)

definition *valid-labeled-moves-max-stone* **where**
valid-labeled-moves-max-stone n l-states ⟷
(∀ *i* < *length l-states* − *1*. *valid-labeled-move-max-stone n* (*l-states* ! *i*) (*l-states*

! (*i* + *1*)))

definition *valid-labeled-game-max-stone* **where**
valid-labeled-game-max-stone n l-states ⟷
length l-states ≥ *2* ∧
hd l-states = *initial-labeled-state n* ∧
last l-states = *final-labeled-state n* ∧
valid-labeled-moves-max-stone n l-states

lemma *valid-labeled-moves-max-stone-Cons*:
assumes *valid-labeled-moves-max-stone n states valid-labeled-move-max-stone n state* (*hd states*)
shows *valid-labeled-moves-max-stone n* (*state* # *states*)
using *assms*
using *less-Suc-eq-0-disj*
apply (*cases states*)
apply (*simp add*: *valid-labeled-moves-max-stone-def*)
apply (*auto simp add*: *valid-labeled-moves-max-stone-def*)
done

lemma *valid-labeled-game-max-stone-valid-labeled-game*:
assumes *valid-labeled-game-max-stone n states*
shows *valid-labeled-game n states*
using *assms*
unfolding *valid-labeled-game-max-stone-def*
unfolding *valid-labeled-game-def valid-labeled-moves-def valid-labeled-moves-max-stone-def*
unfolding *valid-labeled-move-max-stone-def valid-labeled-move-def*
by *force*

lemma *valid-labeled-move-move-max-stone*:
assumes *valid-labeled-state n l-state*
unlabel l-state = *state valid-move′ n p1 p2 state state′*
l-state′ = *labeled-move p1 p2* (*Max* (*l-state* ! *p1*)) *l-state*
shows *valid-labeled-move′ n p1 p2* (*Max* (*l-state* ! *p1*)) *l-state l-state′*
proof−
have *Max* (*l-state* ! *p1*) ∈ *l-state* ! *p1*
by (*metis* (*no-types*, *lifting*) *Max-in assms*(*1*) *assms*(*2*) *assms*(*3*) *card-empty card-infinite less-le-trans nat-neq-iff nth-map trans-less-add1 unlabel-def valid-labeled-state-def valid-move′-def*)
then show *?thesis*

```
    using assms
    by (metis (no-types, lifting) less-le-trans nth-map trans-less-add1 unlabel-def
valid-labeled-move'-def valid-labeled-state-def valid-move'-def)
qed

primrec label-moves-max-stone where
  label-moves-max-stone l-state [] = [l-state]
| label-moves-max-stone l-state (state' # states) =
    (let state = unlabel l-state;
        (p1, p2) = move-positions state state';
        l-state' = labeled-move p1 p2 (Max (l-state ! p1)) l-state
      in l-state # label-moves-max-stone l-state' states)

lemma hd-label-moves-max-stone [simp]:
  shows hd (label-moves-max-stone l-state states) = l-state
  by (induction states) (auto simp add: Let-def split: prod.split)

lemma valid-states-label-moves-max-stone:
  assumes valid-labeled-state n l-state valid-moves n (unlabel l-state # states)
  shows ∀ l-state' ∈ set (label-moves-max-stone l-state states). valid-labeled-state
n l-state'
  using assms
proof (induction states arbitrary: l-state)
  case Nil
  then show ?case
    by simp
next
  case (Cons state' states)
  let ?state = unlabel l-state
  let ?p = move-positions ?state state'
  let ?p1 = fst ?p
  let ?p2 = snd ?p
  let ?stone = Max (l-state ! ?p1)
  let ?l-state' = labeled-move ?p1 ?p2 ?stone l-state

  have valid-state n ?state
    using ‹valid-labeled-state n l-state›
    by (simp add: unlabel-valid)

  have valid-move n ?state state'
```

```
    using Cons(3)
      by (metis Groups.add-ac(2) One-nat-def add-diff-cancel-left' add-is-0 gr0I
list.size(4) n-not-Suc-n nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc valid-moves-def)

  have valid-move' n ?p1 ?p2 ?state state'
    using ⟨valid-state n ?state⟩ ⟨valid-move n ?state state'⟩
    by (simp add: move-positions-valid-move')

  have **: valid-labeled-move' n ?p1 ?p2 ?stone l-state ?l-state'
  proof (rule valid-labeled-move-move-max-stone)
    show valid-labeled-state n l-state
      by fact
  next
    show unlabel l-state = unlabel l-state
      by simp
  next
    show valid-move' n ?p1 ?p2 ?state state'
      by fact
  qed simp

  have move ?p1 ?p2 ?state = state'
    using ⟨valid-move' n ?p1 ?p2 ?state state'⟩
    unfolding valid-move'-def Let-def
    by simp
  then have *: unlabel ?l-state' = state'
    using unlabel-valid-move'[OF Cons(2) **, THEN conjunct2]
    by simp

  have ∀ l-state' ∈ set (label-moves-max-stone ?l-state' states). valid-labeled-state
n l-state'
  proof (rule Cons(1))
    have valid-labeled-move n l-state ?l-state'
      using **
      unfolding valid-labeled-move-def
      by metis
    then show valid-labeled-state n ?l-state'
      using Cons(2)
      using valid-labeled-move-valid-labeled-state
      by blast
  next
```

```
    show valid-moves n (unlabel ?l-state' # states)
      using Cons(3) ⟨valid-move n (unlabel l-state) state'⟩
      using ∗
        by (metis (no-types, lifting) One-nat-def add-Suc-right diff-add-inverse2
group-cancel.add1 less-diff-conv list.size(4) nth-Cons-Suc plus-1-eq-Suc valid-moves-def)
  qed
  then show ?case
    using Cons(2)
     by (metis (mono-tags, lifting) label-moves-max-stone.simps(2) prod.collapse
prod.simps(2) set-ConsD)
qed

lemma unlabel-label-moves-max-stone:
  assumes valid-labeled-state n l-state  valid-moves n (unlabel l-state # states)
   shows map unlabel (label-moves-max-stone l-state states) = unlabel l-state #
states
  using assms
proof (induction states arbitrary: l-state)
  case Nil
  then show ?case
    by simp
next
  case (Cons state' states)
  let ?state = unlabel l-state
  let ?p = move-positions ?state state'
  let ?p1 = fst ?p
  let ?p2 = snd ?p
  let ?stone = Max (l-state ! ?p1)
  let ?l-state' = labeled-move ?p1 ?p2 ?stone l-state

  have valid-state n ?state
    using ⟨valid-labeled-state n l-state⟩
    by (simp add: unlabel-valid)

  have valid-move n ?state state'
    using Cons(3)
      by (metis Groups.add-ac(2) One-nat-def add-diff-cancel-left' add-is-0 gr0I
list.size(4) n-not-Suc-n nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc valid-moves-def)

  have valid-move' n ?p1 ?p2 ?state state'
```

```
    using ⟨valid-state n ?state⟩ ⟨valid-move n ?state state'⟩
    by (simp add: move-positions-valid-move')

  have **: valid-labeled-move' n ?p1 ?p2 ?stone l-state ?l-state'
  proof (rule valid-labeled-move-move-max-stone)
    show valid-labeled-state n l-state
      by fact
  next
    show unlabel l-state = unlabel l-state
      by simp
  next
    show valid-move' n ?p1 ?p2 ?state state'
      by fact
  qed simp

  have move ?p1 ?p2 ?state = state'
    using ⟨valid-move' n ?p1 ?p2 ?state state'⟩
    unfolding valid-move'-def Let-def
    by simp
  then have *: unlabel ?l-state' = state'
    using unlabel-valid-move'[OF Cons(2) **, THEN conjunct2]
    by simp

  have map unlabel (label-moves-max-stone ?l-state' states) = unlabel ?l-state' #
states
  proof (rule Cons(1))
    show valid-moves n ((unlabel ?l-state') # states)
      using Cons(3) *
      using less-diff-conv valid-moves-def
      by auto
  next
    have valid-labeled-move n l-state ?l-state'
      using **
      unfolding valid-labeled-move-def
      by metis
    then show valid-labeled-state n ?l-state'
      using Cons(2)
      using valid-labeled-move-valid-labeled-state
      by blast
  qed
```

```
  then show ?case
    using * ⟨valid-move' n ?p1 ?p2 (unlabel l-state) state'⟩ ⟨valid-state n (unlabel
l-state)⟩
   by (smt Cons-eq-map-conv case-prod-conv label-moves-max-stone.simps(2) valid-move'-move-positions)
qed

lemma label-moves-max-stone-length [simp]:
  shows length (label-moves-max-stone l-state states) = length states + 1
  by (induction states arbitrary: l-state) (auto split: prod.split)

lemma label-moves-max-stone-valid-moves:
  assumes valid-labeled-state n l-state valid-moves n (unlabel l-state # states)
  shows valid-labeled-moves-max-stone n (label-moves-max-stone l-state states)
  using assms
proof (induction states arbitrary: l-state)
  case Nil
  then show ?case
    by (simp add: valid-labeled-moves-max-stone-def)
next
  case (Cons state' states)
  let ?state = unlabel l-state
  let ?p = move-positions ?state state'
  let ?p1 = fst ?p
  let ?p2 = snd ?p
  let ?stone = Max (l-state ! ?p1)
  let ?l-state' = labeled-move ?p1 ?p2 ?stone l-state

  have valid-state n ?state
    using ⟨valid-labeled-state n l-state⟩
    by (simp add: unlabel-valid)

  have valid-move n ?state state'
    using Cons(3)
     by (metis Groups.add-ac(2) One-nat-def add-diff-cancel-left' add-is-0 gr0I
list.size(4) n-not-Suc-n nth-Cons-0 nth-Cons-Suc plus-1-eq-Suc valid-moves-def)

  have valid-move' n ?p1 ?p2 ?state state'
    using ⟨valid-state n ?state⟩ ⟨valid-move n ?state state'⟩
    by (simp add: move-positions-valid-move')
```

```
  have **: valid-labeled-move' n ?p1 ?p2 ?stone l-state ?l-state'
  proof (rule valid-labeled-move-move-max-stone)
    show valid-labeled-state n l-state
      by fact
  next
    show unlabel l-state = unlabel l-state
      by simp
  next
    show valid-move' n ?p1 ?p2 ?state state'
      by fact
  qed simp

  have move ?p1 ?p2 ?state = state'
    using ⟨valid-move' n ?p1 ?p2 ?state state'⟩
    unfolding valid-move'-def Let-def
    by simp
  then have *: unlabel ?l-state' = state'
    using unlabel-valid-move'[OF Cons(2) **, THEN conjunct2]
    by simp

  have ***: valid-labeled-move-max-stone n l-state ?l-state'
    using **
    unfolding valid-labeled-move-max-stone-def
    by blast

  have valid-labeled-moves-max-stone n (label-moves-max-stone ?l-state' states)
  proof (rule Cons(1))
    show valid-moves n ((unlabel ?l-state') # states)
      using Cons(3) *
      using less-diff-conv valid-moves-def
      by auto
    have valid-labeled-move n l-state ?l-state'
      using **
      unfolding valid-labeled-move-def
      by metis
    then show valid-labeled-state n ?l-state'
      using Cons(2)
      using valid-labeled-move-valid-labeled-state
      by blast
  qed
```

```
  moreover
  have hd (label-moves-max-stone ?l-state' states) = ?l-state'
    using hd-label-moves-max-stone by blast
  ultimately
  show ?case
    using *** ⟨valid-move' n ?p1 ?p2 ?state state'⟩ ⟨valid-state n ?state⟩
    using valid-labeled-moves-max-stone-Cons
    by (metis (mono-tags, lifting) case-prod-conv label-moves-max-stone.simps(2)
valid-move'-move-positions)
qed

lemma final-labeled-state-unique:
  assumes unlabel l-state = final-state n valid-labeled-state n l-state
  shows l-state = final-labeled-state n
proof−
  have ∀ i ≤ n. finite (l-state ! i)
     by (metis Groups.add-ac(2) Union-upper assms(2) finite-atLeastLessThan
finite-subset le-imp-less-Suc nth-mem plus-1-eq-Suc valid-labeled-state-def)
  moreover
  have ∀ i < n. card (l-state ! i) = 0
    using assms
    unfolding unlabel-def final-state-def valid-labeled-state-def
   by (metis One-nat-def add.right-neutral add-Suc-right le-imp-less-Suc less-imp-le-nat
nat-neq-iff nth-list-update-neq nth-map nth-replicate)
  moreover
  have card (l-state ! n) = n
    using assms
    unfolding unlabel-def final-state-def valid-labeled-state-def
   by (metis length-replicate less-add-same-cancel1 less-one nth-list-update-eq nth-map)
  moreover
  have ⋃ (set l-state) = {0..<n} length l-state = n + 1
    using assms
    unfolding unlabel-def final-state-def valid-labeled-state-def
    by simp-all
  ultimately
  have ∀ i < n. l-state ! i = {} l-state ! n = {0..<n}
     apply −
     apply auto[1]
   apply (metis Union-upper assms(2) card-atLeastLessThan card-subset-eq diff-zero
finite-atLeastLessThan less-add-same-cancel1 nth-mem valid-labeled-state-def zero-less-one)
```

```
    done
  show ?thesis
  proof (rule nth-equalityI)
    show length l-state = length (final-labeled-state n)
      using ⟨length l-state = n + 1⟩
      unfolding final-labeled-state-def
      by (simp del: replicate-Suc)
  next
    fix i
    assume i < length l-state
    then show l-state ! i = final-labeled-state n ! i
      using ⟨∀ i < n. l-state ! i = {}⟩ ⟨l-state ! n = {0..<n}⟩ ⟨length l-state = n
+ 1⟩
      unfolding final-labeled-state-def
    by (metis add.commute length-replicate less-Suc-eq nth-list-update-eq nth-list-update-neq
nth-replicate plus-1-eq-Suc)
  qed
qed

lemma labeled-game-max-stone-length [simp]:
  assumes valid-game n states
  shows length (label-moves-max-stone (initial-labeled-state n) (tl states)) = length
states
  by (metis assms hd-Cons-tl length-map list.size(3) not-le unlabel-initial unlabel-label-moves-max-ston
valid-game-def valid-labeled-state-initial-labeled-state zero-less-numeral)

lemma valid-labeled-game-max-stone:
  assumes valid-game n states
  shows valid-labeled-game-max-stone n (label-moves-max-stone (initial-labeled-state
n) (tl states))
  unfolding valid-labeled-game-max-stone-def
proof safe
  let ?l-states = label-moves-max-stone (initial-labeled-state n) (tl states)
  have valid-moves n (unlabel (initial-labeled-state n) # tl states)
    using assms
   by (metis Groups.add-ac(2) One-nat-def add-diff-cancel-left' hd-Cons-tl list.sel(2)
list.size(3) list.size(4) n-not-Suc-n plus-1-eq-Suc unlabel-initial upt-0 upt-rec valid-game-def
valid-moves-def)
  then have *: map unlabel ?l-states = (initial-state n) # tl states
    using unlabel-label-moves-max-stone[of n initial-labeled-state n tl states]
```

```
    by simp

  have unlabel (hd ?l-states) = initial-state n
    using ∗
    by auto
  then show hd ?l-states = initial-labeled-state n
    by simp

  have unlabel (last ?l-states) = final-state n
    using assms
    unfolding valid-game-def
   by (metis ∗ Nil-is-map-conv hd-Cons-tl last-map list.size(3) not-le zero-less-numeral)
  moreover
  have valid-labeled-state n (last ?l-states)
    using ∗ ‹valid-moves n (unlabel (initial-labeled-state n) # tl states)›
   by (metis last-in-set list.discI list.simps(8) valid-labeled-state-initial-labeled-state
valid-states-label-moves-max-stone)
  ultimately
  show last ?l-states = final-labeled-state n
    using final-labeled-state-unique
    by blast

 show valid-labeled-moves-max-stone n (label-moves-max-stone (initial-labeled-state
n) (tl states))
  proof (rule label-moves-max-stone-valid-moves)
    show valid-labeled-state n (initial-labeled-state n)
      by simp
  next
    show valid-moves n (unlabel (initial-labeled-state n) # tl states)
      by fact
  qed
next
  show 2 ≤ length (label-moves-max-stone (initial-labeled-state n) (tl states))
    using assms
    unfolding valid-game-def
    by auto
qed
```

Valid labeled game move max stone length

lemma *moved-from*:
  assumes *valid-labeled-state n* (*hd l-states*) *valid-labeled-moves n l-states*
        $i < j$ $j <$ *length l-states stone* $< n$
        stone-position (*l-states* ! i) *stone* $\neq$ *stone-position* (*l-states* ! j) *stone*
  shows ($\exists$ k. $i \leq k \wedge k < j \wedge$
      (*let* (*p1*, *p2*, *stone'*) = *labeled-move-positions* (*l-states* ! k) (*l-states* ! (k + *1*)) *in*
        stone' = *stone* $\wedge$ *p1* = *stone-position* (*l-states* ! i) *stone*))
  using *assms*
proof (*induction l-states arbitrary*: i j)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons l-state l-states*)
  obtain *p1 p2 stone'* **where**
    $*$: (*p1*, *p2*, *stone'*) = *labeled-move-positions* ((*l-state* # *l-states*) ! i) ((*l-state* # *l-states*) ! (i + *1*))
    by (*metis prod-cases3*)

  moreover

  have $***$: *valid-labeled-state n* ((*l-state* # *l-states*) ! i)
    using *Cons*(*2*−*5*)
   by (*meson less-imp-le-nat less-le-trans nth-mem valid-labeled-moves-valid-labeled-states*)

  moreover

  have *valid-labeled-move n* ((*l-state* # *l-states*) ! i) ((*l-state* # *l-states*) ! (i + *1*))
    using *Cons*(*3*−*5*)
    unfolding *valid-labeled-moves-def*
    by *auto*

  ultimately

  have $**$: *valid-labeled-move' n p1 p2 stone'* ((*l-state* # *l-states*) ! i) ((*l-state* # *l-states*) ! (i + *1*))
    using *labeled-move-positions-valid-move'*

```
    by simp

  show ?case
  proof (cases stone' = stone)
    case True
    have p1 = stone-position ((l-state # l-states) ! i) stone'
      using **
    using ⟨valid-labeled-state n ((l-state # l-states) ! i)⟩ valid-labeled-move'-stone-positions
      by blast
    then show ?thesis
      using * Cons(4) True
      by (rule-tac x=i in exI, auto)
  next
    case False

     have stone-position ((l-state # l-states) ! (i + 1)) stone = stone-position
((l-state # l-states) ! i) stone
       using valid-labeled-move'-stone-positions-other[OF *** **] ⟨stone' ≠ stone⟩
⟨stone < n⟩
       by auto
    then have *: stone-position (l-states ! i) stone ≠ stone-position (l-states ! (j
- 1)) stone
      using Cons(4) Cons(7)
      by auto
    moreover
    have valid-labeled-state n (hd l-states)
    proof−
      have l-states ≠ []
        using Cons(4) Cons(5) *
        by auto
      then show ?thesis
        using Cons(2−3)
      by (meson hd-in-set list.set-intros(2) valid-labeled-moves-valid-labeled-states)
    qed

    moreover
    have valid-labeled-moves n l-states
      using Cons(3)
      using Groups.add-ac(2) less-diff-conv valid-labeled-moves-def
      by auto
```

```
    moreover
    have i < j − 1
      using Cons(4) ∗
      using less-antisym plus-1-eq-Suc
      by fastforce
    moreover
    have j − 1 < length l-states
      using ⟨i < j⟩ Cons(5)
      by auto
    ultimately
    obtain k where i ≤ k k < j − 1
        let (p1, p2, stone′) = labeled-move-positions (l-states ! k) (l-states ! (k +
1)) in
          stone′ = stone ∧ p1 = stone-position (l-states ! i) stone
      using Cons(1)[of i j−1] ⟨stone < n⟩
      by (auto simp add: nth-Cons)
    then show ?thesis
      using ⟨stone-position ((l-state # l-states) ! (i + 1)) stone = stone-position
((l-state # l-states) ! i) stone⟩
      by (rule-tac x=k+1 in exI) (auto simp add: Let-def nth-Cons)
  qed
qed

lemma valid-labeled-game-max-stone-min-length:
  assumes valid-labeled-game-max-stone n l-states
  shows length l-states ≥ (∑ k ← [1..<n+1]. (ceiling (n / k))) + 1
  using assms
proof−
  have l-states ≠ [] length l-states ≥ 2 valid-labeled-state n (hd l-states) valid-labeled-moves
n l-states
    using assms
    using valid-labeled-game-max-stone-def
    using valid-labeled-game-def valid-labeled-game-max-stone-valid-labeled-game
    by auto

  let ?ss = map (λ (state, state′). (state, labeled-move-positions state state′)) (zip
(butlast l-states) (tl l-states))
  let ?sstone = λ stone. filter (λ (state, p1, p2, stone′). stone′ = stone) ?ss

  have (∑ k ← [1..<n+1]. (ceiling (n / k))) =
```

```
      (∑ stone ← [0..<n]. (ceiling (n / (stone + 1))))
  proof−
    have map (λx. x + 1) [0..<n] = [1..<n+1]
      using map-add-upt by blast
    then show ?thesis
      by (subst sum-list-comp, simp)
  qed
  also have ... ≤ (∑ stone ← [0..<n]. int (length (?sstone stone)))
  proof (rule sum-list-mono)
    fix stone
    assume stone ∈ set [0..<n]
    show ceiling (n / (stone + 1)) ≤ int (length (?sstone stone))
    proof (rule ccontr)
      assume ¬ ?thesis
      then have ceiling (n / (stone + 1)) > int (length (?sstone stone))
        by simp
      then have int (length (?sstone stone)) * (stone + 1) < n
        using lt-ceiling-frac
        by simp
      then have length (?sstone stone) * (stone + 1) < n
        by (metis (mono-tags, lifting) of-nat-less-imp-less of-nat-mult)

      obtain ss where ss: ss = ?sstone stone
        by auto

      have valid-moves': ∀ (state, p1, p2, stone') ∈ set ss. stone' = Max (state !
p1) ∧ (∃ state'. valid-labeled-move' n p1 p2 stone' state state')
      proof safe
        fix state p1 p2 stone'
        assume (state, p1, p2, stone') ∈ set ss
        then have (state, p1, p2, stone') ∈ set ?ss
          using ss
          by auto
        then obtain state' where
          (state, p1, p2, stone') = (state, labeled-move-positions state state')
          (state, state') ∈ set (zip (butlast l-states) (tl l-states))
          by auto
        then obtain i where i < length ((zip (butlast l-states) (tl l-states))) (zip
(butlast l-states) (tl l-states)) ! i = (state, state')
          by (meson in-set-conv-nth)
```

```
      then have i < length l-states − 1 l-states ! i = state l-states ! (i + 1) =
state'
        using nth-butlast[of i l-states] nth-tl[of i l-states]
        by simp-all
      then have valid-labeled-move-max-stone n state state'
        using ‹valid-labeled-game-max-stone n l-states›
      unfolding valid-labeled-game-max-stone-def valid-labeled-moves-max-stone-def
        by auto
      moreover
      have valid-labeled-state n state
        using ‹i < length l-states − 1› ‹l-states ! i = state›
      by (meson add-lessD1 assms(1) less-diff-conv nth-mem valid-labeled-game-max-stone-valid-labele
valid-labeled-game-valid-labeled-states)
      ultimately
      have ∗: valid-labeled-move' n p1 p2 (Max (state ! p1)) state state'
        using labeled-move-positions valid-labeled-move-max-stone-def
       using ‹(state, p1, p2, stone') = (state, labeled-move-positions state state')›
        by auto

      show stone' = Max (state ! p1)
       using ‹(state, p1, p2, stone') = (state, labeled-move-positions state state')›
‹valid-labeled-move' n p1 p2 (Max (state ! p1)) state state'› ‹valid-labeled-state n
state› labeled-move-positions by auto

      then show (∃ state'. valid-labeled-move' n p1 p2 stone' state state')
        using ∗
        by blast
    qed

    have pos0: stone-position (l-states ! 0) stone = 0
      using ‹stone ∈ set [0..<n]› ‹l-states ≠ []›
      using ‹valid-labeled-game-max-stone n l-states›
      using stone-positionI[of n l-states ! 0 stone 0]
      using hd-conv-nth[of l-states, symmetric]
      using valid-labeled-state-initial-labeled-state
      unfolding valid-labeled-game-max-stone-def initial-labeled-state-def
      by auto

    have posn: stone-position (l-states ! (length l-states − 1)) stone = n
      using stone-positionI[of n l-states ! (length l-states − 1) stone n]
```

```
    using ⟨stone ∈ set [0..<n]⟩ ⟨l-states ≠ []⟩
    using ⟨valid-labeled-game-max-stone n l-states⟩
    using last-conv-nth[of l-states, symmetric]
    using valid-labeled-state-final-labeled-state
    unfolding valid-labeled-game-max-stone-def final-labeled-state-def
    by (simp del: replicate-Suc)

  have n > 0
    using ⟨length (?sstone stone) * (stone + 1) < n⟩ gr-implies-not0
    by blast

  have length ss ≥ 1
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ?sstone stone = []
      using ss
      by (metis One-nat-def Suc-leI length-greater-0-conv)

    have valid-labeled-moves n l-states
      using ⟨valid-labeled-game-max-stone n l-states⟩
      unfolding valid-labeled-game-max-stone-def
   using assms valid-labeled-game-def valid-labeled-game-max-stone-valid-labeled-game
      by blast

    then obtain p2 k where k < length l-states − 1
        (0, p2, stone) = labeled-move-positions (l-states ! k) (l-states ! (k + 1))
      using moved-from[of n l-states 0 length l-states − 1 stone]
      using pos0 posn ⟨n > 0⟩ ⟨stone ∈ set [0..<n]⟩
      using ⟨valid-labeled-game-max-stone n l-states⟩
      unfolding valid-labeled-game-max-stone-def
      by force
    moreover
   have (l-states ! k, l-states ! (k+1)) ∈ set (zip (butlast l-states) (tl l-states))
      using ⟨k < length l-states − 1⟩ ⟨length l-states ≥ 2⟩
       by (metis (no-types, lifting) One-nat-def add.right-neutral add-Suc-right
in-set-conv-nth length-butlast length-tl length-zip min-less-iff-conj nth-butlast nth-tl
nth-zip)
    ultimately
    have (l-states ! k, 0, p2, stone) ∈ set (?sstone stone)
      by auto
```

```
    then show False
      using ‹?sstone stone = []›
      by auto
  qed
  then have ss ≠ []
    by auto

  have n = (∑ (state, p1, p2, stone) ← ?sstone stone. p2 − p1)
  proof−
    let ?p2p1 = λ i. case ss ! i of (state, p1, p2, stone) ⇒ int p2 − int p1
    let ?p1 = λ i. case ss ! i of (state, p1, p2, stone) ⇒ int p1
    let ?p2 = λ i. case ss ! i of (state, p1, p2, stone) ⇒ int p2

    have (∑ (state, p1, p2, stone) ← ss. p2 − p1) =
        (∑ (state, p1, p2, stone) ← ss. int (p2 − p1))
    proof−
      have (∑ (state, p1, p2, stone) ← ss. p2 − p1) =
          (∑ x ← map (λ (state, p1, p2, stone). p2 − p1) ss. int x)
        by (metis (no-types) map-nth sum-list-comp sum-list-int)
      also have ... = (∑(state, p1, p2, stone)←ss. int (p2 − p1))
      proof−
        have ∗: (map int (map (λ (state, p1, p2, stone). p2 − p1) ss)) =
              (map (λ (state, p1, p2, stone). int (p2 − p1)) ss)
          by auto
        show ?thesis
          by (subst ∗, simp)
      qed
      finally show ?thesis
        .
    qed
    also have ... = (∑ (state, p1, p2, stone) ← ss. int p2 − int p1)
    proof−
      have ∀ (state, p1, p2, stone) ∈ set ss. p2 ≥ p1
        using valid-moves′
        unfolding valid-labeled-move′-def Let-def
        by auto
      then have ∀ (state, p1, p2, stone) ∈ set ss. int (p2 − p1) = int p2 −
int p1
        by auto
      then have ∗: map (λ (state, p1, p2, stone). int (p2 − p1)) ss =
```

```
            map (λ (state, p1, p2, stone). int p2 − int p1) ss
        by auto
      show ?thesis
        by (subst ∗, simp)
    qed
    also have ... = (∑ i ← [0..<length ss]. ?p2p1 i)
      by (metis (no-types) map-nth sum-list-comp)
    also have ... = (∑ i ← [0..<length ss]. ?p2 i) −
                    (∑ i ← [0..<length ss]. ?p1 i)
    proof−
      have ∀ i ∈ set [0..<length ss]. ?p2p1 i = ?p2 i − ?p1 i
        by (auto split: prod.split)
         then have map ?p2p1 [0..<length ss] = map (λ i. ?p2 i − ?p1 i)
[0..<length ss]
        by auto
      then show ?thesis
        unfolding Let-def
        by (subst sum-list-subtractf[symmetric], presburger)
    qed
    also have ... = (∑ i ← [0..<length ss−1]. ?p2 i) −
                    (∑ i ← [1..<length ss]. ?p1 i) + (?p2 (length ss−1)) − (?p1
0)
    proof−
      have [0..<length ss] = [0..<length ss−1] @ [length ss − 1]
          [0..<length ss] = [0] @ [1..<length ss]
        using ‹length ss ≥ 1›
        by (metis le-add-diff-inverse plus-1-eq-Suc upt-Suc-append zero-le,
        metis (mono-tags, lifting) One-nat-def le-add-diff-inverse less-numeral-extra(4)
upt-add-eq-append upt-rec zero-le-one zero-less-one)
      then show ?thesis
        using sum-list-append
        by (smt list.map(1) list.map(2) map-append sum-list-simps(2))
    qed
    finally
    have (∑ (state, p1, p2, stone) ← ss. p2 − p1) =
          (∑ i ← [0..<length ss−1]. ?p2 i) −
          (∑ i ← [1..<length ss]. ?p1 i) + (?p2 (length ss−1)) − (?p1 0)
      .
    moreover
```

```
      let ?P = λ(state, p1, p2, stone'). stone' = stone

      have (∑ i ← [1..<length ss]. ?p1 i) = (∑ i ← [0..<length ss − 1]. ?p2 i)
       proof−
        have ∗: ∀ i. 0 < i ∧ i < length ss ⟶ ?p1 i = ?p2 (i−1)
        proof safe
          fix i
          assume 0 < i i < length ss
          show ?p1 i = ?p2 (i−1)
          proof (rule ccontr)
            assume ¬ ?thesis
            obtain k1 k2 where
              k1 < k2 k2 < length ?ss
              ss ! (i−1) = ?ss ! k1 ss ! i = ?ss ! k2
              ?P (?ss ! k1) ?P (?ss ! k2) ∀ k'. k1 < k' ∧ k' < k2 ⟶ ¬ ?P (?ss
! k')
              using ss inside-filter[of i−1 ?P ?ss] ⟨0 < i⟩ ⟨i < length ss⟩
              using ⟨ss ≠ []⟩ ⟨length l-states ≥ 2⟩
              by force
            have k2 < length l-states
              using ⟨k2 < length ?ss⟩
              by simp
             have ?ss ! k1 = (l-states ! k1, labeled-move-positions (l-states ! k1)
(l-states ! (k1+1)))
                  ?ss ! k2 = (l-states ! k2, labeled-move-positions (l-states ! k2)
(l-states ! (k2+1)))
              using ⟨k1 < k2⟩ ⟨k2 < length ?ss⟩ ⟨length l-states ≥ 2⟩
              by (auto simp add: nth-butlast nth-tl)
            then obtain p1a p2a p1b p2b where
                ?ss ! k1 = (l-states ! k1, p1a, p2a, stone) labeled-move-positions
(l-states ! k1) (l-states ! (k1+1)) = (p1a, p2a, stone)
                 ?ss ! k2 = (l-states ! k2, p1b, p2b, stone) labeled-move-positions
(l-states ! k2) (l-states ! (k2+1)) = (p1b, p2b, stone)
              using ⟨?P (?ss ! k1)⟩ ⟨?P (?ss ! k2)⟩
              by auto
            then have p2a ≠ p1b
             using ⟨?p1 i ≠ ?p2 (i−1)⟩ ⟨ss ! (i−1) = ?ss ! k1⟩ ⟨ss ! i = ?ss ! k2⟩
              by simp

          have stone-position (l-states ! (k1 + 1)) stone ≠ stone-position (l-states
```

```
! k2) stone
            proof−
              have valid-labeled-state n (l-states ! k1)
                 by (meson ⟨k1 < k2⟩ ⟨k2 < length l-states⟩ assms less-imp-le-nat
less-le-trans nth-mem valid-labeled-game-max-stone-valid-labeled-game valid-labeled-game-valid-labeled-states)
              moreover
             then have valid-labeled-move′ n p1a p2a stone (l-states ! k1) (l-states
! (k1+1))
               using ⟨labeled-move-positions (l-states ! k1) (l-states ! (k1+1)) =
(p1a, p2a, stone)⟩
              using labeled-move-positions-valid-move′
             using ⟨k1 < k2⟩ ⟨k2 < length l-states⟩ ⟨valid-labeled-moves n l-states⟩
valid-labeled-moves-def
              by auto
             ultimately
             have stone-position (l-states ! (k1 + 1)) stone = p2a
              using valid-labeled-move′-stone-positions
              by blast

             have valid-labeled-state n (l-states ! k2)
              by (meson ⟨k2 < length l-states⟩ assms less-imp-le-nat less-le-trans
nth-mem valid-labeled-game-max-stone-valid-labeled-game valid-labeled-game-valid-labeled-states)
              moreover
             then have valid-labeled-move′ n p1b p2b stone (l-states ! k2) (l-states
! (k2+1))
               using ⟨labeled-move-positions (l-states ! k2) (l-states ! (k2+1)) =
(p1b, p2b, stone)⟩
              using labeled-move-positions-valid-move′
                    using ⟨k2 < length ?ss⟩ ⟨valid-labeled-moves n l-states⟩
valid-labeled-moves-def
            by (smt length-butlast length-map length-tl length-zip min-less-iff-conj)
             ultimately
             have stone-position (l-states ! k2) stone = p1b
              using valid-labeled-move′-stone-positions
              by blast

             show ?thesis
              using ⟨stone-position (l-states ! k2) stone = p1b⟩
              using ⟨stone-position (l-states ! (k1 + 1)) stone = p2a⟩
              using ⟨p2a ≠ p1b⟩
```

```
            by simp
          qed
          then have k1 + 1 < k2
            using ⟨k1 < k2⟩
            by (metis Suc-eq-plus1 Suc-leI nat-less-le)
          then obtain k' p1'' p2'' where k1 + 1 ≤ k' k' < k2
            (p1'', p2'', stone) = labeled-move-positions (l-states ! k') (l-states !
(k' + 1))
             using ⟨stone-position (l-states ! (k1 + 1)) stone ≠ stone-position
(l-states ! k2) stone⟩
            using moved-from[of n l-states k1+1 k2 stone] ⟨stone ∈ set [0..<n]⟩
            using ⟨length l-states ≥ 2⟩ ⟨k1 < k2⟩ ⟨k2 < length l-states⟩
          using ⟨valid-labeled-moves n l-states⟩ ⟨valid-labeled-state n (hd l-states)⟩
            by auto
          then have ?ss ! k' = (l-states ! k', p1'', p2'', stone)
            using ⟨k2 < length ?ss⟩
            by (auto simp add: nth-butlast nth-tl)
          then show False
            using ⟨∀ k'. k1 < k' ∧ k' < k2 ⟶ ¬ ?P (?ss ! k')⟩[rule-format, of
k'] ⟨k1 + 1 ≤ k'⟩ ⟨k' < k2⟩
            by simp
        qed
      qed
      have map ?p1 [1..<length ss] = map ?p2 [0..<length ss − 1] (is ?lhs =
?rhs)
      proof (rule nth-equalityI)
        show length ?lhs = length ?rhs
          by simp
      next
        fix i
        assume i < length ?lhs
        then show ?lhs ! i = ?rhs ! i
          using ∗
          by simp
      qed
      then show ?thesis
        by simp
    qed
    moreover
    have ?p2 (length ss − 1) = n
```

proof (*rule ccontr*)
assume ¬ *?thesis*
obtain *k* **where**
k < *length ?ss ss* ! (*length ss* − *1*) = *?ss* ! *k ?P* (*?ss* ! *k*) ∀ *k′*. *k* < *k′* ∧ *k′* < *length ?ss* ⟶ ¬ *?P* (*?ss* ! *k′*)
using *ss last-filter*[*of ?P ?ss*]
using ‹*ss* ≠ []› ‹*length l-states* ≥ *2*›
by *auto*
have *k* < *length l-states* − *1*
using ‹*k* < *length ?ss*›
by *simp*
have *?ss* ! *k* = (*l-states* ! *k*, *labeled-move-positions* (*l-states* ! *k*) (*l-states* ! (*k*+*1*)))
using ‹*k* < *length ?ss*› ‹*length l-states* ≥ *2*›
by (*auto simp add*: *nth-butlast nth-tl*)
then obtain *p1′ p2′* **where** *?ss* ! *k* = (*l-states* ! *k*, *p1′*, *p2′*, *stone*) *labeled-move-positions* (*l-states* ! *k*) (*l-states* ! (*k*+*1*)) = (*p1′*, *p2′*, *stone*)
using ‹*?P* (*?ss* ! *k*)›
by *auto*
then have *p2′* ≠ *n*
using ‹*?p2* (*length ss* − *1*) ≠ *n*› ‹*ss* ! (*length ss* − *1*) = *?ss* ! *k*›
by *auto*
have *stone-position* (*l-states* ! (*k* + *1*)) *stone* ≠ *n*
proof−
have *stone-position* (*l-states* ! (*k* + *1*)) *stone* = *p2′*
proof−
have *valid-labeled-move′ n p1′ p2′ stone* (*l-states* ! *k*) (*l-states* ! (*k*+*1*))
using ‹*labeled-move-positions* (*l-states* ! *k*) (*l-states* ! (*k*+*1*)) = (*p1′*, *p2′*, *stone*)›
using ‹*k* < *length l-states* − *1*› ‹*valid-labeled-moves n l-states*› ‹*valid-labeled-state n* (*hd l-states*)›
by (*meson add-lessD1 labeled-move-positions-valid-move′ less-diff-conv nth-mem valid-labeled-moves-def valid-labeled-moves-valid-labeled-states*)
moreover
have *valid-labeled-state n* (*l-states* ! *k*)
using ‹*k* < *length l-states* − *1*› ‹*valid-labeled-moves n l-states*› ‹*valid-labeled-state n* (*hd l-states*)›
using *valid-labeled-moves-valid-labeled-states*
by *auto*
ultimately

```
        show ?thesis
          using valid-labeled-move'-stone-positions
          by blast
      qed
      then show ?thesis
        using ⟨p2' ≠ n⟩
        by simp
    qed
    then have k + 1 < length l-states − 1
      using posn ⟨k < length l-states − 1⟩
     by (smt Nat.le-diff-conv2 Nat.le-imp-diff-is-add Suc-leI add.right-neutral
add-Suc-right add-leD2 diff-diff-left nat-less-le one-add-one plus-1-eq-Suc)
   then obtain k' p1'' p2'' where k' ≥ k + 1 k' < length l-states − 1 (p1'',
p2'', stone) = labeled-move-positions (l-states ! k') (l-states ! (k' + 1))
      using moved-from[of n l-states k+1 length l-states − 1 stone]
       using posn ⟨stone-position (l-states ! (k+1)) stone ≠ n⟩ ⟨stone ∈ set
[0..<n]⟩
      using ⟨length l-states ≥ 2⟩
      using ⟨valid-labeled-moves n l-states⟩ ⟨valid-labeled-state n (hd l-states)⟩
      by force
    then have ?ss ! k' = (l-states ! k', p1'', p2'', stone)
      by (simp add: nth-butlast nth-tl)
    then show False
      using ⟨∀ k'. k < k' ∧ k' < length ?ss ⟶ ¬ ?P (?ss ! k')⟩[rule-format,
of k'] ⟨k' ≥ k + 1⟩ ⟨k' < length l-states − 1⟩
      by auto
  qed
  moreover
  have ?p1 0 = 0
  proof (rule ccontr)
    assume ¬ ?thesis
    obtain k where
      k < length ?ss ss ! 0 = ?ss ! k ?P (?ss ! k) ∀ k' < k. ¬ ?P (?ss ! k')
      using ss hd-filter[of ?P ?ss]
      using ⟨ss ≠ []⟩ ⟨length l-states ≥ 2⟩
      by auto
    have k < length l-states − 1
      using ⟨k < length ?ss⟩
      by simp
    have ?ss ! k = (l-states ! k, labeled-move-positions (l-states ! k) (l-states
```

```
! (k+1)))
          using ‹k < length ?ss› ‹length l-states ≥ 2›
          by (auto simp add: nth-butlast nth-tl)
           then obtain p1' p2' where ?ss ! k = (l-states ! k, p1', p2', stone)
labeled-move-positions (l-states ! k) (l-states ! (k+1)) = (p1', p2', stone)
          using ‹?P (?ss ! k)›
          by auto
        then have p1' ≠ 0
          using ‹?p1 0 ≠ 0› ‹ss ! 0 = ?ss ! k›
          by auto
        have stone-position (l-states ! k) stone ≠ 0
        proof−
          have valid-labeled-state n (l-states ! k)
            by (meson ‹k < length l-states − 1› add-lessD1 assms less-diff-conv
nth-mem valid-labeled-game-max-stone-valid-labeled-game valid-labeled-game-valid-labeled-states)
          moreover
           then have valid-labeled-move' n p1' p2' stone (l-states ! k) (l-states !
(k+1))
            using ‹labeled-move-positions (l-states ! k) (l-states ! (k+1)) = (p1',
p2', stone)›
              using ‹k < length l-states − 1› ‹valid-labeled-moves n l-states›
labeled-move-positions-valid-move' valid-labeled-moves-def
            by blast
          ultimately
          have stone-position (l-states ! k) stone = p1'
            using valid-labeled-move'-stone-positions
            by blast
          then show ?thesis
            using ‹p1' ≠ 0›
            by simp
        qed
        then have k > 0
          using pos0
          using neq0-conv
          by blast
        have k < length l-states
          using ‹k < length l-states − 1› ‹length l-states ≥ 2›
          by auto
         then obtain k' p2'' where k' < k labeled-move-positions (l-states ! k')
(l-states ! (k' + 1)) = (0, p2'', stone)
```

```
        using moved-from[of n l-states 0 k stone] pos0 ‹stone-position (l-states !
k) stone ≠ 0›
        using ‹valid-labeled-state n (hd l-states)› ‹valid-labeled-moves n l-states›
‹k > 0› ‹stone ∈ set [0..<n]›
        by auto
      then have ?ss ! k' = (l-states ! k', 0, p2'', stone)
        using ‹k' < k› ‹k < length l-states − 1›
        using ‹k < length ?ss› ‹length l-states ≥ 2›
        by (auto simp add: nth-butlast nth-tl)
      then show False
        using ‹∀ k' < k. ¬ ?P (?ss ! k')›[rule-format, of k'] ‹k' < k›
        by simp
    qed
    ultimately
    show ?thesis
      using ss
      by simp
  qed
  also have ... ≤ (∑ (state, p1, p2, stone) ← ?sstone stone. stone + 1)
  proof (rule sum-list-mono)
    fix x :: labeled-state × nat × nat × nat
    obtain state p1 p2 stone' where x: x = (state, p1, p2, stone')
      by (cases x)
    assume x ∈ set (?sstone stone)
    then have x ∈ set ss
      using ss
      by auto
    then obtain state' where stone' = Max (state ! p1) valid-labeled-move' n
p1 p2 (Max (state ! p1)) state state'
      using x valid-moves'
      by auto
    then have p1 < p2 p2 ≤ p1 + card (state ! p1)
      unfolding valid-labeled-move'-def Let-def
      by auto

    moreover

    have card (state ! p1) ≤ Max (state ! p1) + 1
      by (rule card-Max)
```

```
      ultimately

      show (case x of (state, p1, p2, stone) ⇒ p2 − p1) ≤
          (case x of (state, p1, p2, stone) ⇒ stone + 1)
        using x ‹stone′ = Max (state ! p1)›
        by simp
    qed
    also have ... = (∑ x ← ?sstone stone. stone + 1)
    proof−
      have map (λ (state, p1, p2, stone). stone + 1) (?sstone stone) =
          map (λ x. stone + 1) (?sstone stone)
        by auto
      then show ?thesis
        by presburger
    qed
    also have ... = length (?sstone stone) ∗ (stone + 1)
      by (simp add: sum-list-triv)
    finally
    show False
      using ‹length (?sstone stone) ∗ (stone + 1) < n›
      by simp
  qed
qed
also have ... ≤ length ?ss
proof−
  let ?ps = map (λ stone. (λ(state, p1, p2, stone′). stone′ = stone)) [0..<n]
  have ∀ i j. i < j ∧ j < length ?ps ⟶ set (filter (?ps ! i) ?ss) ∩ set (filter
(?ps ! j) ?ss) = {}
    by auto
  then have (∑ stone ← [0..<n]. length (?sstone stone)) ≤ length ?ss
    using sum-length-parts[of ?ps ?ss]
    by (auto simp add: comp-def split: prod.split)
  then show ?thesis
    by (subst sum-list-int, simp)
qed
finally
have (∑ k ← [1..<n+1]. (ceiling (n / k))) + 1 ≤ length ?ss + 1
  by simp
moreover
have length ?ss + 1 = length l-states
```

```
    using ⟨l-states ≠ []⟩
    by simp
  ultimately
  show ?thesis
    by simp
qed
```

Valid game length

```
theorem IMO2018SL-C3:
  assumes valid-game n states
  shows length states ≥ (∑ k ← [1..<n+1]. (ceiling (n / k))) + 1
proof−
  let ?l-states = label-moves-max-stone (initial-labeled-state n) (tl states)
  have length ?l-states = length states
    using assms
    unfolding valid-game-def
    by auto
  moreover
  have valid-labeled-game-max-stone n ?l-states
    using valid-labeled-game-max-stone[OF assms]
    by simp
  ultimately
  show ?thesis
    using valid-labeled-game-max-stone-min-length[of n ?l-states]
    by simp
qed

end
```

5.2.4 IMO 2018 SL - C4

```
theory IMO-2018-SL-C4-sol
  imports Main HOL−Library.Permutation
begin

definition antipascal :: (nat ⇒ nat ⇒ int) ⇒ nat ⇒ bool where
    antipascal f n ⟷ (∀ r < n. ∀ c ≤ r. f r c = abs (f (r+1) c − f (r+1)
(c+1)))
```

definition *triangle* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* × *nat*) *set* **where**
triangle r0 c0 n = {(*r*, *c*) | *r c* :: *nat*. *r0* ≤ *r* ∧ *r* < *r0* + *n* ∧ *c0* ≤ *c* ∧ *c* ≤ *c0* + *r* − *r0*}

lemma *triangle-finite* [*simp*]:
shows *finite* (*triangle r0 c0 n*)
proof−
have *triangle r0 c0 n* ⊆ {*0*..<*r0* + *n*} × {*0*..<*c0* + *n*}
unfolding *triangle-def*
by *auto*
then show *?thesis*
using *finite-atLeastLessThan infinite-super*
by *blast*
qed

lemma *triangle-card*:
shows *card* (*triangle r0 c0 n*) = *n* ∗ (*n*+*1*) *div 2*
proof (*induction n arbitrary*: *r0 c0*)
case *0*
have ∗: {(*i*, *j*). *r0* ≤ *i* ∧ *i* < *r0* ∧ *c0* ≤ *j* ∧ *j* ≤ *c0* + *i* − *r0*} = {}
by *auto*
then show *?case*
using *0*
unfolding *triangle-def*
by (*simp add*: ∗)
next
case (*Suc n*)
let *?row* = {(*r0* + *n*, *j*) | *j*. *c0* ≤ *j* ∧ *j* < *c0* + *Suc n*}
have *triangle r0 c0* (*Suc n*) = *triangle r0 c0 n* ∪ *?row*
unfolding *triangle-def*
by *auto*
moreover
have *triangle r0 c0 n* ∩ *?row* = {}
unfolding *triangle-def*
by *auto*
ultimately
have *card* (*triangle r0 c0* (*Suc n*)) = *card* (*triangle r0 c0 n*) + *card ?row*
by (*simp add*: *card-Un-disjoint*)
moreover
have *card ?row* = *Suc n*

```
  proof−
    have ?row = (λ j. (r0 + n, j)) ‘ {c0..<c0 + Suc n}
      by auto
    moreover
    have inj-on (λ j. (r0 + n, j)) {c0..<c0 + Suc n}
      unfolding inj-on-def
      by auto
    then have card ((λ j. (r0 + n, j)) ‘ {c0..<c0 + Suc n}) = card {c0..<c0 +
Suc n}
      using card-image
      by blast
    then have card ((λ j. (r0 + n, j)) ‘ {c0..<c0 + Suc n}) = Suc n
      by auto
    ultimately
    show ?thesis
      by simp
  qed
  ultimately
  have card (triangle r0 c0 (Suc n)) = (n ∗ (n + 1)) div 2 + Suc n
    using Suc
    by simp
  then show ?case
    by auto
qed

fun uncurry where
   uncurry f (a, b) = f a b

lemma gauss:
  fixes n :: nat
  shows sum-list [1..<n] = n ∗ (n − 1) div 2
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have sum-list [1..<Suc n] = sum-list [1..<n] + n
    by simp
  also have ... = n ∗ (n − 1) div 2 + n
    using Suc
```

```
    by simp
  finally
  show ?case
     by (metis Sum-Ico-nat diff-self-eq-0 distinct-sum-list-conv-Sum distinct-upt
minus-nat.diff-0 mult-eq-0-iff set-upt)
qed

lemma sum-list-insort [simp]:
  fixes x :: nat and xs :: nat list
  shows sum-list (insort x xs) = x + sum-list xs
  by (induction xs, auto)

lemma sum-list-sort [simp]:
  fixes xs :: nat list
  shows sum-list (sort xs) = sum-list xs
  by (induction xs, auto)

lemma sorted-distinct-strict-increase:
  assumes sorted (xs @ [x]) distinct (xs @ [x]) ∀ x ∈ set (xs @ [x]). x > a
  shows x > a + length xs
  using assms
proof (induction xs arbitrary: x rule: rev-induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc x' xs)
  show ?case
    using snoc(1)[of x'] snoc(2−)
    by (auto simp add: sorted-append)
qed

lemma sum-list-sorted-distinct-lb:
  assumes ∀ x ∈ set xs. x > a distinct xs sorted xs
  shows sum-list xs ≥ length xs * (2 * a + length xs + 1) div 2
  using assms
proof (induction xs rule: rev-induct)
  case Nil
  then show ?case
    by simp
```

```
next
  case (snoc x xs)
  have x > a + length xs
    using sorted-distinct-strict-increase[of xs x a]
    using snoc(2−)
    by auto
  moreover
  have length xs * (2 * a + length xs + 1) div 2 ≤ sum-list xs
    using snoc
    by (auto simp add: sorted-append)
  ultimately
  show ?case
    by auto
qed

lemma sum-list-distinct-lb:
  assumes ∀ x ∈ set xs. f x > a distinct (map f xs)
  shows (∑ x ← xs. f x) ≥ length xs * (2 * a + length xs + 1) div 2
  using assms
  using sum-list-sorted-distinct-lb[of sort (map f xs) a]
  by simp

lemma consecutive-nats-sorted:
  assumes sorted xs length xs = n distinct xs sum-list xs ≤ n * (n + 1) div 2 ∀
x ∈ set xs. x > 0
  shows xs = [1..<n+1]
  using assms
proof (induction xs arbitrary: n rule: rev-induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc x xs)
  have n > 0
    using ‹length (xs @ [x]) = n›
    by simp
  have xs = [1..<(n−1)+1]
  proof (rule snoc(1))
    show sorted xs length xs = n−1 distinct xs ∀ a∈set xs. 0 < a
      using snoc(2−6)
```

```
    by (auto simp add: sorted-append)
  show sum-list xs ≤ (n − 1) ∗ (n − 1 + 1) div 2
  proof−
    have x ≥ n
      using snoc(2−4) snoc(6)
    proof (induction xs arbitrary: x n rule: rev-induct)
      case Nil
      then show ?case
        by simp
    next
      case (snoc x′ xs′)
      have n−1 ≤ x′
        using snoc(1)[of x′ n−1] snoc(2−)
        by (simp add: sorted-append)
      moreover
      have x > x′
        using snoc(2) snoc(4)
        by (simp add: sorted-append)
      ultimately
      show ?case
        by simp
    qed
    show ?thesis
    proof−
      have sum-list xs ≤ n ∗ (n + 1) div 2 − x
        using snoc(5)
        by simp
      also have ... ≤ n ∗ (n + 1) div 2 − n
        using ‹n ≤ x›
        by simp
      also have ... = n ∗ (n − 1) div 2
        by (simp add: diff-mult-distrib2)
      finally
      show ?thesis
        using ‹n > 0›
        by (auto simp add: mult.commute)
    qed
  qed
qed
then have xs = [1..<n]
```

```
    using ‹n > 0›
    by simp
  then have x ≥ n
    using snoc(2) snoc(4) snoc(6)
    by (auto simp add: sorted-append)
  have x = n
  proof (rule ccontr)
    assume ¬ ?thesis
    then have x > n
      using ‹x ≥ n›
      by simp
    then have sum-list (xs @ [x]) > n * (n - 1) div 2 + n
      using ‹xs = [1..<n]› gauss[of n]
      by simp
    then show False
      using snoc(5)
    by (smt Suc-diff-1 ‹0 < n› add.commute add-Suc-right distrib-left div-mult-self2
less-le-trans mult-2 mult-2-right nat-neq-iff one-add-one plus-1-eq-Suc zero-neq-numeral)
  qed
  then show ?case
    using ‹xs = [1..<n]›
    by (simp add: Suc-leI ‹0 < n›)
qed

lemma consecutive-nats:
  assumes length xs = n distinct xs sum-list xs ≤ n * (n + 1) div 2 ∀ x ∈ set
xs. x > 0
  shows set xs = {1..<n+1}
proof−
  have sort xs = [1..<n+1]
    using consecutive-nats-sorted[of sort xs n] assms
    by simp
  then show ?thesis
    by (metis set-sort set-upt)
qed

lemma sum-list-cong:
  assumes ∀ x ∈ set xs. f x = g x
  shows (∑ x ← xs. f x) = (∑ x ← xs. g x)
  using assms
```

```
  by (induction xs, auto)

lemma sum-list-last:
  assumes a ≤ b
  shows (∑ x ← [a..<b+1]. f x) = (∑ x ← [a..<b]. f x) + f b
proof−
  have ∗: [a..<b+1] = [a..<b] @ [b]
    using assms
    by auto
  show ?thesis
    by (subst ∗, simp)
qed

lemma sum-list-nat:
  assumes ∀ x ∈ set xs. f x ≥ 0
  shows (∑ x ← xs. nat (f x)) = (nat (∑ x ← xs. f x))
  using assms
proof (induction xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  then show ?case
    using sum-list-mono
    by fastforce
qed

theorem IMO2018SL-C4:
  ∄ f. antipascal f 2018 ∧
   (uncurry f) ‘ triangle 0 0 2018 = {1..<2018∗(2018 + 1) div 2 + 1}
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain f where
   f: antipascal f 2018 (uncurry f) ‘ triangle 0 0 2018 = {1..<2018∗(2018 + 1)
div 2+1}
    by auto

  have inj-on (uncurry f) (triangle 0 0 2018)
  proof (rule eq-card-imp-inj-on)
```

```
    show finite (triangle 0 0 2018)
      by simp
  next
    show card ((uncurry f) ` triangle 0 0 2018) = card (triangle 0 0 2018)
      using f(2) triangle-card
      by simp
  qed

  have path: ∀ r0 < 2018. ∀ c0 ≤ r0. ∀ n. r0 + n ≤ 2018 ⟶ (∃ a b. a r0 =
c0 ∧ b r0 = c0 ∧
                  (∀ r. r0 < r ∧ r < r0 + n ⟶
                      a r ≠ b r ∧ c0 ≤ a r ∧ a r ≤ c0 + (r − r0) ∧ c0 ≤ b r ∧ b
r ≤ c0 + (r − r0) ∧
                        (a r = b (r − 1) ∨ a r = b (r − 1) + 1) ∧
                        (b r = b (r − 1) ∨ b r = b (r − 1) + 1)) ∧
                  (∀ r. r0 ≤ r ∧ r < r0 + n ⟶ f r (b r) = (∑ r' ← [r0..<r+1].
f r' (a r')))) (is ∀ r0 < 2018. ∀ c0 ≤ r0. ∀ n. r0 + n ≤ 2018 ⟶ ?P r0 c0 n)
  proof safe
    fix r0 c0 n :: nat
    assume r0 < 2018 c0 ≤ r0 r0 + n ≤ 2018
    then show ?P r0 c0 n
    proof (induction n)
      case 0
      then show ?case
        by auto
    next
      case (Suc n)

      show ?case
      proof (cases n = 0)
        case True
        then show ?thesis
          by auto
      next
        case False
        show ?thesis
        proof−
          obtain a b where ∗:
            a r0 = c0 b r0 = c0
            ∀ r. r0 < r ∧ r < r0 + n ⟶
```

```
        a r ≠ b r ∧
        c0 ≤ a r ∧ a r ≤ c0 + (r − r0) ∧
        c0 ≤ b r ∧ b r ≤ c0 + (r − r0) ∧
        (a r = b (r − 1) ∨ a r = b (r − 1) + 1) ∧
        (b r = b (r − 1) ∨ b r = b (r − 1) + 1)
    ∀r. r0 ≤ r ∧ r < r0 + n ⟶ f r (b r) = (∑ r'←[r0..<r + 1]. f r' (a r'))
    using Suc
    by auto

  have ap': ∀r c. r0 ≤ r ∧ r ≤ r0 + n ∧ c0 ≤ c ∧ c < c0 + (r − r0)
⟶ f (r − 1) c = |f r c − f r (c + 1)|
    using ⟨antipascal f 2018⟩ ⟨n ≠ 0⟩ Suc(3−4)
    unfolding antipascal-def
    by auto
  have ap: f (r0 + n − 1) (b (r0 + n − 1)) = |f (r0 + n) (b (r0 + n −
1)) − f (r0 + n) (b (r0 + n − 1) + 1)|
  proof (cases n = 1)
    case True
    then show ?thesis
      using ∗(2) ap'[rule-format, of r0 + 1]
      by simp
  next
    case False
    then have n > 1
      using ⟨n ≠ 0⟩
      by simp
    show ?thesis
    proof (subst ap')
      have r0 < r0 + n − 1
        using ⟨n > 1⟩
        by simp
      then have b (r0 + n − Suc 0) ≤ c0 + n − Suc 0
        using ∗(3)[rule-format, of r0 + n − 1] ⟨n > 1⟩
        by simp
     then show r0 ≤ r0 + n ∧ r0 + n ≤ r0 + n ∧ c0 ≤ b (r0 + n − 1)
∧ b (r0 + n − 1) < c0 + (r0 + n − r0)
        using ∗(3)[rule-format, of r0 + n − 1] ⟨n > 1⟩
        by simp
    qed simp
```

```
    qed

    let ?an = if f (r0 + n) (b (r0 + n − 1)) < f (r0 + n) (b (r0 + n −
1) + 1) then b (r0 + n − 1) else b (r0 + n − 1) + 1
    let ?bn = if f (r0 + n) (b (r0 + n − 1)) < f (r0 + n) (b (r0 + n − 1)
+ 1) then b (r0 + n − 1) + 1 else b (r0 + n − 1)
    let ?a = a (r0 + n := ?an)
    let ?b = b (r0 + n := ?bn)

    have ?a r0 = c0 ?b r0 = c0
      using ⟨n ≠ 0⟩ ⟨a r0 = c0⟩ ⟨b r0 = c0⟩
      by simp-all

    moreover

    have ∀ r. r0 ≤ r ∧ r < r0 + Suc n ⟶ f r (?b r) = (∑ r′ ← [r0..<r+1].
f r′ (?a r′))
    proof safe
      fix r
      assume r0 ≤ r r < r0 + Suc n
      show f r (?b r) = (∑ r′ ← [r0..<r+1]. f r′ (?a r′))
      proof (cases r < r0 + n)
        case True
        then have f r (?b r) = (∑ r′ ← [r0..<r+1]. f r′ (a r′))
          using ∗(4) ⟨r0 ≤ r⟩
          by simp
        also have ... = (∑ r′ ← [r0..<r+1]. f r′ (?a r′))
        proof (rule sum-list-cong, safe)
          fix r′
          assume r′ ∈ set [r0..<r + 1]
          then show f r′ (a r′) = f r′ (?a r′)
            using True ⟨r0 ≤ r⟩
            by auto
        qed
        finally show ?thesis
          by simp
      next
        case False
        then have r = r0 + n
          using ⟨r < r0 + Suc n⟩
```

by *simp*
show *?thesis*
proof (*cases f (r0 + n) (b (r0 + n − 1)) < f (r0 + n) (b (r0 + n − 1) + 1)*)
case *True*
have *f (r0 + n) (b (r0 + n − 1) + 1) = f (r0 + n − 1) (b (r0 + n − 1)) + f (r0 + n) (b (r0 + n − 1))*
using *True ap*
by *simp*
then have *f (r0 + n) (b (r0 + n − 1) + 1) = ((∑ r′←[r0..<r0 + n]. f r′ (a r′))) + f (r0 + n) (b (r0 + n − 1))*
using ∗(*4*) ‹*n ≠ 0*›
by *simp*
also have *... = (∑ r′←[r0..<r0 + n]. f r′ (if r′ = r0 + n then b (r0 + n − 1) else (a r′))) + f (r0 + n) (if r0 + n = r0 + n then b (r0 + n − 1) else (a (r0 + n)))*
proof−
have *(∑ r′←[r0..<r0 + n]. f r′ (a r′)) = (∑ r′←[r0..<r0 + n]. f r′ (if r′ = r0 + n then b (r0 + n − 1) else (a r′)))*
by (*rule sum-list-cong*, *simp*)
then show *?thesis*
by *simp*
qed
also have *... = (∑ r′←[r0..<r0 + n + 1]. f r′ (if r′ = r0 + n then b (r0 + n − 1) else (a r′)))*
by (*subst sum-list-last*, *simp-all*)
finally show *?thesis*
using *True* ‹*r = r0 + n*›
by *simp* (*metis One-nat-def*)
next
case *False*
then have *f (r0 + n) (b (r0 + n − 1)) = f (r0 + n − 1) (b (r0 + n − 1)) + f (r0 + n) (b (r0 + n − 1) + 1)*
using *ap*
by *simp*
then have *f (r0 + n) (b (r0 + n − 1)) = ((∑ r′←[r0..<r0 + n]. f r′ (a r′))) + f (r0 + n) (b (r0 + n − 1) + 1)*
using ∗(*4*) ‹*n ≠ 0*›
by *simp*
also have *... = (∑ r′←[r0..<r0 + n]. f r′ (if r′ = r0 + n then b (r0*

```
+ n − 1) + 1 else (a r′))) + f (r0 + n) (if r0 + n = r0 + n then b (r0 + n −
1) + 1 else (a (r0 + n)))
            proof−
              have (∑ r′←[r0..<r0 + n]. f r′ (a r′)) = (∑ r′←[r0..<r0 + n]. f
r′ (if r′ = r0 + n then b (r0 + n − 1) + 1 else (a r′)))
                by (rule sum-list-cong, simp)
              then show ?thesis
                by simp
            qed
            also have ... = (∑ r′←[r0..<r0 + n + 1]. f r′ (if r′ = r0 + n then
b (r0 + n − 1) + 1 else (a r′)))
              by (subst sum-list-last, simp-all)
            finally
            show ?thesis
              using False ‹r = r0 + n›
              by simp (metis One-nat-def Suc-eq-plus1)
          qed
        qed
      qed

      moreover

      have ∀ r. r0 < r ∧ r < r0 + Suc n ⟶
            ?a r ≠ ?b r ∧
            c0 ≤ ?a r ∧ ?a r ≤ c0 + (r − r0) ∧
            c0 ≤ ?b r ∧ ?b r ≤ c0 + (r − r0) ∧
            (?a r = ?b (r − 1) ∨ ?a r = ?b (r − 1) + 1) ∧
            (?b r = ?b (r − 1) ∨ ?b r = ?b (r − 1) + 1)
      proof safe
        fix r
        assume r0 < r r < r0 + Suc n ?a r = ?b r
        then show False
          using ∗
          by (simp split: if-split-asm)
      next
        fix r
        assume r0 < r r < r0 + Suc n
        show c0 ≤ ?a r
        proof (cases r < r0 + n)
          case True
```

```
        then show ?thesis
          using * ⟨r0 < r⟩
          by auto
      next
        case False
        then have r = r0 + n
          using ⟨r < r0 + Suc n⟩
          by simp
        then show ?thesis
          using *(2) *(3)[rule-format, of r0 + n - 1]
        by (smt Suc-diff-1 Suc-eq-plus1 Suc-leD Suc-le-mono ⟨r0 < r⟩ add-gr-0
diff-less fun-upd-same less-antisym less-or-eq-imp-le zero-less-one)
      qed
    next
      fix r
      assume r0 < r r < r0 + Suc n
      show ?a r ≤ c0 + (r - r0)
      proof (cases r < r0 + n)
        case True
        then show ?thesis
          using * ⟨r0 < r⟩
          by auto
      next
        case False
        then have r = r0 + n
          using ⟨r < r0 + Suc n⟩
          by simp
        then show ?thesis
          using *(2) *(3)[rule-format, of r0 + n - 1]
            by (smt Suc-diff-Suc ⟨r0 < r⟩ add-Suc-right add-diff-cancel-left'
add-diff-cancel-right' fun-upd-same le-Suc-eq less-Suc-eq less-or-eq-imp-le nat-add-left-cancel-le
plus-1-eq-Suc)
      qed
    next
      fix r
      assume r0 < r r < r0 + Suc n
      show c0 ≤ ?b r
      proof (cases r < r0 + n)
        case True
        then show ?thesis
```

```
        using * ⟨r0 < r⟩
        by auto
      next
        case False
        then have r = r0 + n
          using ⟨r < r0 + Suc n⟩
          by simp
        then show ?thesis
          using *(2) *(3)[rule-format, of r0 + n - 1]
        by (smt Suc-diff-1 Suc-eq-plus1 Suc-leD Suc-le-mono ⟨r0 < r⟩ add-gr-0
diff-less fun-upd-same less-antisym less-or-eq-imp-le zero-less-one)
      qed
    next
      fix r
      assume r0 < r r < r0 + Suc n
      show ?b r ≤ c0 + (r - r0)
      proof (cases r < r0 + n)
        case True
        then show ?thesis
          using * ⟨r0 < r⟩
          by auto
      next
        case False
        then have r = r0 + n
          using ⟨r < r0 + Suc n⟩
          by simp
        then show ?thesis
          using *(2) *(3)[rule-format, of r0 + n - 1]
            by (smt Suc-diff-Suc ⟨r0 < r⟩ add-Suc-right add-diff-cancel-left'
add-diff-cancel-right' fun-upd-same le-Suc-eq less-Suc-eq less-or-eq-imp-le nat-add-left-cancel-le
plus-1-eq-Suc)
      qed
    next
      fix r
      assume r0 < r r < r0 + Suc n
           ?a r ≠ ?b (r - 1) + 1
      then show ?a r = ?b (r - 1)
        using *
        by (auto split: if-split-asm)
    next
```

```
        fix r
        assume r0 < r r < r0 + Suc n
               ?b r ≠ ?b (r − 1) + 1
        then show ?b r = ?b (r − 1)
          using ∗
          by (auto split: if-split-asm)
      qed
      ultimately
      show ?thesis
        by blast
    qed
  qed
qed

obtain a b where ∗:
  a 0 = 0 b 0 = 0
  ∀ r. 0 < r ∧ r < 2018 ⟶ a r ≠ b r
  ∀ r. 0 < r ∧ r < 2018 ⟶ a r ≤ r
  ∀ r. 0 < r ∧ r < 2018 ⟶ b r ≤ r
  ∀ r. 0 < r ∧ r < 2018 ⟶ a r = b (r − 1) ∨ a r = b (r − 1) + 1
  ∀ r. 0 < r ∧ r < 2018 ⟶ b r = b (r − 1) ∨ b r = b (r − 1) + 1
  ∀ r<2018. f r (b r) = (∑ r' ← [0..<r+1]. f r' (a r'))
  using path[rule-format, of 0 0 2018]
  by auto

have ab: ∀ r < 2018. a r = b r + 1 ∨ a r = b r − 1
  using ∗(1−3) ∗(6−7)
 by (metis Suc-eq-plus1 diff-add-inverse diff-is-0-eq' le0 neq0-conv plus-1-eq-Suc)

have max-max: ∀ r. 0 < r ∧ r < 2018 ⟶ max (a (r − 1)) (b (r − 1)) ≤ max
(a r) (b r)
proof safe
  fix r :: nat
  assume r: 0 < r r < 2018
  show max (a (r − 1)) (b (r − 1)) ≤ max (a r) (b r)
  proof (cases r = 1)
    case True
    then show ?thesis
      using ⟨a 0 = 0⟩ ⟨b 0 = 0⟩
```

```
      by simp
    next
      case False
      then have a r = b (r − 1) ∨ a r = b (r − 1) + 1
                b r = b (r − 1) ∨ b r = b (r − 1) + 1
                a r ≠ b r a (r − 1) ≠ b (r − 1)
        using r ∗
        by simp-all
      moreover
      have a (r − 1) = b (r − 1) ∨ a (r − 1) = b (r − 1) + 1 ∨ a (r − 1) =
b (r − 1) − 1
        using ab[rule-format, of r−1] r False
        by auto
      ultimately
      show ?thesis
        by (smt diff-le-self eq-iff le-add1 max.commute max.mono)
    qed
  qed

  have min-min: ∀ r. 0 < r ∧ r < 2018 ⟶ min (a (r − 1)) (b (r − 1)) ≥ min
(a r) (b r) − 1
    using ∗(2) ∗(3) ∗(6) ∗(7)
   by (smt One-nat-def Suc-diff-Suc Suc-leD cancel-comm-monoid-add-class.diff-cancel
diff-zero le-0-eq le-diff-conv less-Suc-eq min.cobounded1 min-def nat-less-le)

  let ?fa = map (λ r. f r (a r)) [0..<2018]

  have inj-on (λ r. f r (a r)) (set [0..<2018])
    unfolding inj-on-def
  proof safe
    fix r1 r2
    assume r1 ∈ set [0..<2018] r2 ∈ set [0..<2018]
      f r1 (a r1) = f r2 (a r2)
    have (r1, a r1) ∈ triangle 0 0 2018 (r2, a r2) ∈ triangle 0 0 2018
      using ‹r1 ∈ set [0..<2018]› ‹r2 ∈ set [0..<2018]› ∗(4) ∗(1)
      using le-eq-less-or-eq triangle-def
      by auto
    moreover
    have f r1 (a r1) = (uncurry f) (r1, a r1)  f r2 (a r2) = (uncurry f) (r2, a
r2)
```

```
    by auto
  ultimately
  show r1 = r2
    using ‹inj-on (uncurry f) (triangle 0 0 2018)› ‹f r1 (a r1) = f r2 (a r2)›
    by (metis inj-onD prod.inject)
qed

have distinct ?fa
  using ‹inj-on (λ r. f r (a r)) (set [0..<2018])›
  by (simp add: distinct-map)

have ∀ x ∈ set ?fa. x > 0
proof safe
  fix x
  assume x ∈ set ?fa
  then obtain r where r < 2018 x = f r (a r)
    by auto

  have (r, a r) ∈ triangle 0 0 2018
    using ∗(4) ∗(1) ‹r < 2018›
    by (cases r = 0, auto simp add: triangle-def)
  moreover
  have (uncurry f) (r, a r) = f r (a r)
    by auto
  ultimately
  have f r (a r) ∈ (uncurry f) ‘ triangle 0 0 2018
    by (metis rev-image-eqI)
  then show x > 0
    using f(2) ‹x = f r (a r)›
    by auto
qed

have set (map nat ?fa) = {1..<2018+1}
proof (rule consecutive-nats)
  show length (map nat ?fa) = 2018
    by simp
next
  show distinct (map nat ?fa)
  proof (subst distinct-map, safe)
    show distinct (map (λr. f r (a r)) [0..<2018])
```

```
      by fact
  next
    show inj-on nat (set ?fa)
      using ⟨∀ x ∈ set ?fa. x > 0⟩ inj-on-def
      by force
  qed
next
  show ∀ x ∈ set (map nat ?fa). x > 0
    using ⟨∀ x ∈ set ?fa. x > 0⟩
    by simp
next
  show sum-list (map nat ?fa) ≤ 2018 * (2018 + 1) div 2
  proof−
    have (∑ x ← ?fa. x) ∈ (uncurry f) ‘ (triangle 0 0 2018)
    proof−
      have (∑ x ← ?fa. x) = f 2017 (b 2017)
        using ∗(8)[rule-format, of 2017]
        by simp
      moreover
      have (2017, b 2017) ∈ triangle 0 0 2018
        using ∗(5)
        unfolding triangle-def
        by simp
      moreover
      have (uncurry f) (2017, b 2017) = f 2017 (b 2017)
        by simp
      ultimately
      show ?thesis
        by force
    qed
    then have (∑ x ← ?fa. x) ≤ 2018∗(2018 + 1) div 2
      using f(2)
      by auto
    moreover
    have ∀ x∈{0..<2018}. 0 ≤ f x (a x)
      by (simp add: ⟨∀ x∈set (map (λr. f r (a r)) [0..<2018]). 0 < x⟩ le-less)
    ultimately
    show ?thesis
      using sum-list-nat[of [0..<2018] (λr. f r (a r))]
      by (simp add: comp-def)
```

```
    qed
  qed

  have ?fa <~~> map int [1..<2018+1]
  proof−
    have set ?fa = set (map int [1..<2018+1])
    proof (subst inj-on-Un-image-eq-iff[symmetric])
      show nat ‘ set ?fa = nat ‘ set (map int [1..<2018+1])
      proof−
        have set (map nat ?fa) = nat ‘ set ?fa
          by auto
        moreover
        have nat ‘ set (map int [1..<2018+1]) = {1..<2018+1}
          by (metis (no-types, hide-lams) atLeastLessThan-upt map-idI map-map
nat-int o-apply set-map)
        ultimately
        show ?thesis
          using ⟨set (map nat ?fa) = {1..<2018+1}⟩
          by simp
      qed
    next
      show inj-on nat (set ?fa ∪ set (map int [1..<2018 + 1]))
      proof−
        have set ?fa ∪ set (map int [1..<2018 + 1]) ⊆ {x::int. x > 0}
          using ⟨∀ x ∈ set ?fa. x > 0⟩
          by auto
        moreover
        have inj-on nat {x::int. x > 0}
          by (metis inj-on-inverseI mem-Collect-eq nat-int zero-less-imp-eq-int)
        ultimately
        show ?thesis
          by (smt inj-onD inj-onI subsetD)
      qed
    qed

    then have mset ?fa = mset (map int [1..<2018+1])
    proof (subst set-eq-iff-mset-eq-distinct[symmetric])
      show distinct ?fa
        by fact
    next
```

```
    show distinct (map int [1..<2018+1])
      by (simp add: distinct-map)
  qed simp

  then show ?thesis
    using mset-eq-perm
    by blast
qed

let ?l = min (a 2017) (b 2017)
let ?r = max (a 2017) (b 2017)
let ?r0l = 2018 - ?l and ?c0l = 0 and ?nl = ?l
let ?r0r = ?r + 1 and ?c0r = ?r + 1 and ?nr = 2017 - ?r

{
  fix r0 c0 n
  assume triangle r0 c0 n ⊆ triangle 0 0 2018
  assume ∀ r < 2018. (r, a r) ∉ triangle r0 c0 n
  assume n ≥ 1008
  assume c0 ≤ r0 r0 + n ≤ 2018

  have ∀ p ∈ triangle r0 c0 n. (uncurry f) p > 2018
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain r c where (r, c) ∈ triangle r0 c0 n f r c ≤ 2018
      by auto
    moreover
    have (r, c) ∈ triangle 0 0 2018
      using ‹triangle r0 c0 n ⊆ triangle 0 0 2018› ‹(r, c) ∈ triangle r0 c0 n›
      by auto
    then have f r c ≥ 1
      using ‹(uncurry f) ‘ (triangle 0 0 2018) = {1..<2018*(2018 + 1) div 2 +
1}›
      by force
    then have nat (f r c) ∈ {1..<2018+1}
      using ‹f r c ≤ 2018›
      by auto
    then have f r c ∈ set (map int [1..<2018+1])
      by (smt ‹1 ≤ f r c› atLeastLessThan-upt image-eqI int-nat-eq set-map)
    then have f r c ∈ set ?fa
```

```
      using ‹?fa <~~> map int [1..<2018+1]›
      using perm-set-eq
      by blast
    then obtain r' where r' < 2018 f r' (a r') = f r c
      by auto
    have (r', a r') ∈ triangle 0 0 2018
      using ‹r' < 2018› *(1) *(4)
      by (cases r' = 0) (auto simp add: triangle-def)
    then have r = r' c = a r'
      using ‹f r' (a r') = f r c› ‹inj-on (uncurry f) (triangle 0 0 2018)› ‹(r, c) ∈
triangle 0 0 2018›
      unfolding inj-on-def
      by force+

    then have (r', a r') ∈ triangle r0 c0 n
      using ‹(r, c) ∈ triangle r0 c0 n›
      by simp

    then show False
      using ‹r' < 2018› ‹∀ r < 2018. (r, a r) ∉ triangle r0 c0 n›
      by auto
  qed

  obtain ar br where r:
    ar r0 = c0 br r0 = c0
    ∀ r. r0 < r ∧ r < r0 + n ⟶ ar r ≠ br r ∧
                c0 ≤ ar r ∧ ar r ≤ c0 + (r − r0) ∧
                c0 ≤ br r ∧ br r ≤ c0 + (r − r0) ∧
                (ar r = br (r − 1) ∨ ar r = (br (r − 1)) + 1) ∧
                (br r = br (r − 1) ∨ br r = (br (r − 1)) + 1)
    ∀ r. r0 ≤ r ∧ r < r0 + n ⟶
                f r (br r) =
                (∑ r'←[r0..<r+1]. f r' (ar r'))
    using ‹r0 + n ≤ 2018› ‹c0 ≤ r0› ‹n ≥ 1008›
    using path[rule-format, of r0 c0 n]
    by auto

  have ∀ r. r0 ≤ r ∧ r < r0 + n ⟶ (r, ar r) ∈ triangle r0 c0 n
  proof safe
    fix r
```

```
    assume r0 ≤ r r < r0 + n
    then show (r, ar r) ∈ triangle r0 c0 n
      using r(1) r(2) r(3)[rule-format, of r]
      unfolding triangle-def
      by (cases r = r0) auto
  qed
  then have ∀ r. r0 ≤ r ∧ r < r0 + n ⟶ f r (ar r) > 2018
    using ⟨∀ p ∈ triangle r0 c0 n. (uncurry f) p > 2018⟩
    by force

  have (r0 + n − 1, br (r0 + n − 1)) ∈ triangle r0 c0 n
    using r(3)[rule-format, of r0 + n − 1]
    using ⟨r0 + n ≤ 2018⟩ ⟨n ≥ 1008⟩
    by (simp add: triangle-def)
  then have (r0 + n − 1, br (r0 + n − 1)) ∈ triangle 0 0 2018
    using ⟨triangle r0 c0 n ⊆ triangle 0 0 2018⟩
    by blast
  then have (uncurry f) (r0 + n − 1, br (r0 + n − 1)) ∈ {1..<2018 ∗ (2018
+ 1) div 2 + 1}
    using f(2)
    by blast
  then have f (r0 + n − 1) (br (r0 + n − 1)) ≤ 2018∗(2018+1) div 2
    by simp

  moreover

  have f (r0 + n − 1) (br (r0 + n − 1)) = (∑ r'←[r0..<(r0 + n − 1) + 1].
f r' (ar r'))
    using r(4)[rule-format, of r0 + n − 1]
    using ⟨r0 + n ≤ 2018⟩ ⟨n ≥ 1008⟩
    by simp

  ultimately

  have 1: (∑ r'←[r0..<(r0 + n − 1) + 1]. f r' (ar r')) ≤ 2018∗(2018+1) div
2
    by simp

  have length ([r0..< (r0 + n − 1) + 1]) = n
    using ⟨n ≥ 1008⟩
```

```
    by auto

  have n * (2 * 2018 + n + 1) div 2 ≥ 1008 * (2*2018 + 1008 + 1) div 2
  proof−
    have n * (2 * 2018 + n + 1) ≥ 1008 * (2*2018 + 1008 + 1)
      using ‹n ≥ 1008›
      by (metis Suc-eq-plus1 add-Suc mult-le-mono nat-add-left-cancel-le)
    then show ?thesis
      using div-le-mono
      by blast
  qed

  moreover

  have length [r0..<(r0 + n − 1) + 1] * (2 * 2018 + length [r0..<(r0 + n −
1) + 1] + 1) div 2 ≤
        (∑ r′←[r0..<(r0 + n − 1) + 1]. nat (f r′ (ar r′)))
  proof (rule sum-list-distinct-lb)
    have ∀ r′∈set [r0..<(r0 + n − 1) + 1]. 2018 < f r′ (ar r′)
      using ‹∀ r. r0 ≤ r ∧ r < r0 + n ⟶ f r (ar r) > 2018› ‹n ≥ 1008›
      by simp
    then show ∀ r′∈set [r0..<(r0 + n − 1)+ 1]. 2018 < nat (f r′ (ar r′))
      by auto
  next
    show distinct (map (λx. nat (f x (ar x))) [r0..<(r0 + n − 1) + 1])
    proof (subst distinct-map, safe)
      show inj-on (λx. nat (f x (ar x))) (set [r0..<(r0 + n − 1) + 1])
        unfolding inj-on-def
      proof safe
        fix r1 r2
        assume r1 ∈ set [r0..<(r0 + n − 1) + 1] r2 ∈ set [r0..<(r0 + n − 1)
+ 1]
               nat (f r1 (ar r1)) = nat (f r2 (ar r2))
        have (r1, ar r1) ∈ triangle r0 c0 n (r2, ar r2) ∈ triangle r0 c0 n
          using ‹r1 ∈ set [r0..<(r0 + n − 1) + 1]› ‹r2 ∈ set [r0..<(r0 + n −
1) + 1]›
          using ‹∀ r. r0 ≤ r ∧ r < r0 + n ⟶ (r, ar r) ∈ triangle r0 c0 n›
          using ‹n ≥ 1008›
          by force+
```

```
      then have (r1, ar r1) ∈ triangle 0 0 2018 (r2, ar r2) ∈ triangle 0 0 2018
        using ⟨triangle r0 c0 n ⊆ triangle 0 0 2018⟩
        by blast+

      moreover

      have f r1 (a r1) = (uncurry f) (r1, a r1)  f r2 (a r2) = (uncurry f) (r2,
a r2)
        by auto

      moreover

      have f r1 (a r1) = f r2 (a r2)
       using ⟨(r1, ar r1) ∈ triangle 0 0 2018⟩ ⟨(r2, ar r2) ∈ triangle 0 0 2018⟩
        using ⟨nat (f r1 (ar r1)) = nat (f r2 (ar r2))⟩
        using ⟨(r1, ar r1) ∈ triangle r0 c0 n⟩
        using ⟨∀ p ∈ triangle r0 c0 n. 2018 < uncurry f p⟩
        using ⟨inj-on (uncurry f) (triangle 0 0 2018)⟩
        by (smt Pair-inject eq-nat-nat-iff inj-on-def nat-0-iff uncurry.simps)

      ultimately

      show r1 = r2
        using ⟨inj-on (uncurry f) (triangle 0 0 2018)⟩
        using ⟨(r1, ar r1) ∈ triangle r0 c0 n⟩
              ⟨∀ p∈triangle r0 c0 n. 2018 < uncurry f p⟩
              ⟨nat (f r1 (ar r1)) = nat (f r2 (ar r2))⟩
        by (smt Pair-inject inj-on-eq-iff int-nat-eq uncurry.simps)
    qed
  qed simp
qed

ultimately

have (∑ r'←[r0..<(r0 + n − 1) + 1]. nat (f r' (ar r'))) ≥ 1008 ∗ (2∗2018
+ 1008 + 1) div 2
  using ⟨length ([r0..< (r0 + n − 1) + 1]) = n⟩
  by simp

moreover
```

```
  have (∑ r'←[r0..<(r0 + n − 1) + 1]. nat (f r' (ar r'))) = nat ((∑ r'←[r0..<(r0
+ n − 1) + 1]. f r' (ar r')))
   proof (rule sum-list-nat)
    show ∀ r'∈set [r0..<(r0 + n − 1) + 1]. 0 ≤ f r' (ar r')
     using ⟨∀ r. r0 ≤ r ∧ r < r0 + n ⟶ f r (ar r) > 2018⟩ ⟨n ≥ 1008⟩
     by auto
   qed

   ultimately

   have 2: nat ((∑ r'←[r0..<(r0 + n − 1) + 1]. f r' (ar r'))) ≥ 1008 * (2*2018
+ 1008 + 1) div 2
    by simp


   have False
    using 1 2
    by simp
 } note triangle = this

 show False
 proof (cases ?nl ≤ ?nr)
  case True
  show False
  proof (rule triangle)
   show triangle ?r0r ?c0r ?nr ⊆ triangle 0 0 2018
    unfolding triangle-def
    by auto
  next
   show ?nr ≥ 1008
    using ab[rule-format, of 2017] True
    by (auto simp add: max-def min-def split: if-split-asm)
  next
   show ∀ r < 2018. (r, a r) ∉ triangle ?r0r ?c0r ?nr
   proof−
    have ∀ r < 2018. max (a r) (b r) ≤ ?r
    proof−
     have ∀ r < 2018. max (a (2017 − r)) (b (2017 − r)) ≤ ?r
     proof safe
```

```
          fix r::nat
          assume r < 2018
          then show max (a (2017 − r)) (b (2017 − r)) ≤ ?r
          proof (induction r)
            case 0
            then show ?case
              by simp
          next
            case (Suc r)
            then show ?case
              using max-max ∗(1−2)
                  by (smt Suc-diff-Suc Suc-lessD add-diff-cancel-left′ diff-Suc-Suc
diff-less-Suc max.boundedE max.orderE one-plus-numeral plus-1-eq-Suc semiring-norm(4)
semiring-norm(5) zero-less-diff)
          qed
        qed
        then show ?thesis
       by (metis Suc-leI add-le-cancel-left diff-diff-cancel diff-less-Suc one-plus-numeral
plus-1-eq-Suc semiring-norm(4) semiring-norm(5))
      qed
      then show ?thesis
        unfolding triangle-def
        by auto
    qed
  next
    show ?r0r ≤ ?c0r
      by simp
  next
    show ?r0r + ?nr ≤ 2018
      by (simp add: ∗(4) ∗(5))
  qed
next
  case False
  show ?thesis
  proof (rule triangle)
    show triangle ?r0l ?c0l ?nl ⊆ triangle 0 0 2018
      using ∗(4)[rule-format, of 2017] ∗(5)[rule-format, of 2017]
      unfolding triangle-def
      by auto
  next
```

```
    show ?c0l ≤ ?r0l
      by simp
  next
    show 2018 − min (a 2017) (b 2017) + min (a 2017) (b 2017) ≤ 2018
      using ∗(4)[rule-format, of 2017] ∗(5)[rule-format, of 2017]
      by auto
  next
    show ?nl ≥ 1008
      using ab[rule-format, of 2017] False
      by (auto simp add: max-def min-def split: if-split-asm)
  next
    show ∀ r < 2018. (r, a r) ∉ triangle ?r0l ?c0l ?nl
    proof−
      have ∀ r < 2018. min (a r) (b r) ≥ ?l − (2017 − r)
      proof−
        have ∀ r < 2018. min (a (2017 − r)) (b (2017 − r)) ≥ ?l − (2017 −
(2017 − r))
        proof safe
          fix r::nat
          assume r < 2018
          then show ?l − (2017 − (2017 − r)) ≤ min (a (2017 − r)) (b (2017
− r))
          proof (induction r)
            case 0
            then show ?case
              by simp
          next
            case (Suc r)
             have min (a 2017) (b 2017) − (2017 − (2017 − Suc r)) = min (a
2017) (b 2017) − r − 1
              using ‹Suc r < 2018›
              by auto
            also have ... ≤ min (a (2017 − r)) (b (2017 − r)) − 1
              using Suc
                by (smt Suc-lessD diff-Suc-Suc diff-diff-cancel diff-le-mono le-less
one-plus-numeral plus-1-eq-Suc semiring-norm(4) semiring-norm(5) zero-less-diff)
            also have ... ≤ min (a (2017 − r − 1)) (b (2017 − r − 1))
              using min-min[rule-format, of 2017 − r] ‹Suc r < 2018›
              by simp
            finally
```

```
            show ?case
              by simp
          qed
        qed
        then show ?thesis
          by (smt diff-diff-cancel diff-less-Suc le-less less-Suc-eq one-plus-numeral
plus-1-eq-Suc semiring-norm(4) semiring-norm(5))
      qed
      then show ?thesis
        by (auto simp add: triangle-def)
    qed
  qed
 qed
qed

end
```

5.3 Number theory problems

5.3.1 IMO 2018 SL - N5

```
theory IMO-2018-SL-N5-sol
imports Main
begin

definition perfect-square :: int ⇒ bool where
  perfect-square s ⟷ (∃ r. s = r * r)

lemma perfect-square-root-pos:
  assumes perfect-square s
  shows ∃ r. r ≥ 0 ∧ s = r * r
  using assms
  unfolding perfect-square-def
  by (smt mult-minus-left mult-minus-right)

lemma not-perfect-square-15:
  fixes q::int
  shows q^2 ≠ 15
proof (rule ccontr)
  assume ¬ ?thesis
```

```
  then have 3^2 < (abs q)^2 (abs q)^2 < 4^2
    by auto
  then have 3 < abs q abs q < 4
    using abs-ge-zero power-less-imp-less-base zero-le-numeral
    by blast+
  then show False
    by simp
qed

lemma not-perfect-square-12:
  fixes q::int
  shows q^2 ≠ 12
proof (rule ccontr)
  assume ¬ ?thesis
  then have 3^2 < (abs q)^2 (abs q)^2 < 4^2
    by auto
  then have 3 < abs q abs q < 4
    using abs-ge-zero power-less-imp-less-base zero-le-numeral
    by blast+
  then show False
    by simp
qed

lemma not-perfect-square-8:
  fixes q::int
  shows q^2 ≠ 8
proof (rule ccontr)
  assume ¬ ?thesis
  then have 2^2 < (abs q)^2 (abs q)^2 < 3^2
    by auto
  then have 2 < abs q abs q < 3
    using abs-ge-zero power-less-imp-less-base zero-le-numeral
    by blast+
  then show False
    by simp
qed

lemma not-perfect-square-7:
  fixes q::int
  shows q^2 ≠ 7
```

proof (*rule ccontr*)
assume ¬ *?thesis*
then have *2^2 < (abs q)^2 (abs q)^2 < 3^2*
by *auto*
then have *2 < abs q abs q < 3*
using *abs-ge-zero power-less-imp-less-base zero-le-numeral*
by *blast+*
then show *False*
by *simp*
qed

lemma *not-perfect-square-5*:
fixes *q*::*int*
shows *q^2 ≠ 5*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *2^2 < (abs q)^2 (abs q)^2 < 3^2*
by *auto*
then have *2 < abs q abs q < 3*
using *abs-ge-zero power-less-imp-less-base zero-le-numeral*
by *blast+*
then show *False*
by *simp*
qed

lemma *not-perfect-square-3*:
fixes *q*::*int*
shows *q^2 ≠ 3*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *1^2 < (abs q)^2 (abs q)^2 < 2^2*
by *auto*
then have *1 < abs q abs q < 2*
using *abs-ge-zero power-less-imp-less-base zero-le-numeral*
by *blast+*
then show *False*
by *simp*
qed

lemma *IMO2018SL-N5-lemma*:

fixes *s a b c d* :: *int*
assumes *s^2 = a^2 + b^2 s^2 = c^2 + d^2 2*s = a^2 − c^2*
assumes *s > 0 a ≥ 0 d ≥ 0 b ≥ 0 c ≥ 0 b > 0 ∨ c > 0 b ≥ c*
shows *False*
proof−
have *2*s = d^2 − b^2*
using *assms*
by *simp*

have *d > 0*
using ⟨*2 ∗ s = d^2 − b^2*⟩ ⟨*s > 0*⟩ ⟨*d ≥ 0*⟩
by (*smt pos-imp-zdiv-neg-iff zero-less-power2*)

have *a > 0*
using ⟨*2 ∗ s = a^2 − c^2*⟩ ⟨*s > 0*⟩ ⟨*a ≥ 0*⟩
by (*smt pos-imp-zdiv-neg-iff zero-less-power2*)

have *b > 0*
using *assms*
by *auto*

have *d^2 > c^2*
using ⟨$2 * s = d^2 - b^2$⟩ ⟨*c ≤ b*⟩ ⟨*0 < s*⟩ ⟨*c ≥ 0*⟩
by (*smt power-mono*)

then have *d^2 > s^2 div 2*
using ⟨*s^2 = c^2 + d^2*⟩
by *presburger*

then have *2*s^2 < 4*d^2*
by *simp*

have *b < d*
using ⟨*2*s = d^2 − b^2*⟩ ⟨*s > 0*⟩ ⟨*d > 0*⟩ ⟨*b > 0*⟩
by (*smt power-mono-iff zero-less-numeral*)

have *even b ⟷ even d*
using ⟨*2*s = d^2 − b^2*⟩
by (*metis add-uminus-conv-diff dvd-minus-iff even-add even-mult-iff even-numeral power2-eq-square*)

```
  then have b ≤ d − 2
    using ⟨b < d⟩
    by (smt even-two-times-div-two odd-two-times-div-two-succ)

  then have 2*s ≥ d^2 − (d−2)^2
    using ⟨2*s = d^2 − b^2⟩ ⟨d > 0⟩ ⟨b > 0⟩
    by auto
  then have s ≥ 2*(d − 1)
    by (simp add: algebra-simps power2-eq-square)
  then have 2*d ≤ s + 2
    by simp
  then have 4*d^2 ≤ (s + 2)^2
    using abs-le-square-iff[of 2*d s + 2] ⟨d > 0⟩ ⟨s > 0⟩
    by auto
  then have 2*s^2 < (s+2)^2
    using ⟨2*s^2 < 4*d^2⟩
    by simp
  then have (s − 2)^2 < 8
    by (simp add: power2-eq-square algebra-simps)
  then have (s − 2)^2 < 3^2
    by simp
  then have s − 2 < 3
    using power2-less-imp-less
    by fastforce
  then have s ≤ 4
    by simp
  then have s = 1 ∨ s = 2 ∨ s = 3 ∨ s = 4
    using ⟨s > 0⟩
    by auto
  moreover
  have ⋀ p q :: int. ⟦16 = p^2 + q^2; p ≥ 0; q ≥ 0⟧ ⟹ p = 0 ∨ q = 0
  proof−
    fix p q :: int
    assume 16 = p^2 + q^2 p ≥ 0 q ≥ 0
    have p ≤ 4
    proof (rule ccontr)
      assume ¬ ?thesis
      then have p ≥ 5
        by simp
```

```
    then have p^2 ≥ 25
      using power-mono[of 5 p 2]
      by simp
    then have p^2 + q^2 ≥ 25
      using zero-le-power2[of q]
      by linarith
    then show False
      using ⟨16 = p^2 + q^2⟩
      by auto
  qed
  then have p = 0 ∨ p = 1 ∨ p = 2 ∨ p = 3 ∨ p = 4
    using ⟨0 ≤ p⟩
    by auto
  then show p = 0 ∨ q = 0
  using ⟨16 = p^2 + q^2⟩ not-perfect-square-15 not-perfect-square-12 not-perfect-square-7
    by auto
qed
moreover
have ⋀ p q :: int. ⟦9 = p^2 + q^2; p ≥ 0; q ≥ 0⟧ ⟹ p = 0 ∨ q = 0
proof−
  fix p q :: int
  assume 9 = p^2 + q^2 p ≥ 0 q ≥ 0
  have p ≤ 3
  proof (rule ccontr)
    assume ¬ ?thesis
    then have p ≥ 4
      by simp
    then have p^2 ≥ 16
      using power-mono[of 4 p 2]
      by simp
    then have p^2 + q^2 ≥ 16
      using zero-le-power2[of q]
      by linarith
    then show False
      using ⟨9 = p^2 + q^2⟩
      by auto
  qed
  then have p = 0 ∨ p = 1 ∨ p = 2 ∨ p = 3
    using ⟨0 ≤ p⟩
    by auto
```

```
  then show p = 0 ∨ q = 0
    using ⟨9 = p^2 + q^2⟩ not-perfect-square-8 not-perfect-square-5
    by auto
qed
moreover
have ⋀ p q :: int. ⟦4 = p^2 + q^2; p ≥ 0; q ≥ 0⟧ ⟹ p = 0 ∨ q = 0
proof−
  fix p q :: int
  assume 4 = p^2 + q^2 p ≥ 0 q ≥ 0
  have p ≤ 2
  proof (rule ccontr)
    assume ¬ ?thesis
    then have p ≥ 3
      by simp
    then have p^2 ≥ 9
      using power-mono[of 3 p 2]
      by simp
    then have p^2 + q^2 ≥ 9
      using zero-le-power2[of q]
      by linarith
    then show False
      using ⟨4 = p^2 + q^2⟩
      by auto
  qed
  then have p = 0 ∨ p = 1 ∨ p = 2
    using ⟨0 ≤ p⟩
    by auto
  then show p = 0 ∨ q = 0
    using ⟨4 = p^2 + q^2⟩ not-perfect-square-3
    by auto
qed
moreover
have ⋀ p q :: int. ⟦1 = p^2 + q^2; p ≥ 0; q ≥ 0⟧ ⟹ p = 0 ∨ q = 0
  by (smt one-le-power)
moreover
have a ≠ 0 d ≠ 0
  using ⟨a > 0⟩ ⟨d > 0⟩
  by auto
ultimately
have c = 0 b = 0
```

```
    using ⟨s^2 = c^2 + d^2⟩ ⟨d ≥ 0⟩ ⟨c ≥ 0⟩ ⟨s^2 = a^2 + b^2⟩ ⟨a ≥ 0⟩ ⟨b ≥ 0⟩
    by fastforce+
  then show False
    using ⟨b > 0 ∨ c > 0⟩
    by auto
qed

theorem IMO2018SL-N5:
  fixes x y z t :: int
  assumes pos: x > 0 y > 0 z > 0 t > 0
  assumes eq: x*y - z*t = x + y x + y = z + t
  shows  ¬ (perfect-square (x*y) ∧ perfect-square (z*t))
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain a c where x*y = a*a z*t = c*c a > 0 c > 0
    using perfect-square-root-pos pos
    by (smt zero-less-mult-iff)

  show False
  proof (cases odd (x + y))
    case True

    have even (x * y)
      using True
      by auto

    moreover

    have odd (z + t)
      using True eq(2)
      by simp
    then have even (z * t)
      by auto

    ultimately

    have even (x*y - z*t)
      by simp
    then show False
      using eq(1) True
```

```
    by simp
  next
    case False
    then have even (x + y) even (z + t)
      using eq(2)
      by auto

    let ?s = (x + y) div 2
    let ?b = abs (x − y) div 2 and ?d = abs (z − t) div 2

    have ?s ^ 2 = a ^ 2 + ?b ^ 2
    proof−
      have a^2 + ?b^2 = (x+y)^2 div 4
        using ⟨even (x+y)⟩ div-power[of 2 abs (x − y) 2] ⟨x∗y = a∗a⟩
        by (simp add: power2-eq-square algebra-simps)
      then show ?thesis
        by (metis False div-power mult-2-right numeral-Bit0 power2-eq-square)
    qed

    have ?s ^ 2 = c ^ 2 + ?d ^ 2
    proof−
      have c^2 + ?d^2 = (z+t)^2 div 4
        using ⟨even (z+t)⟩ div-power[of 2 abs (z − t) 2] ⟨z∗t = c∗c⟩
        by (simp add: power2-eq-square algebra-simps)
      then show ?thesis
       by (metis eq(2) False div-power mult-2-right numeral-Bit0 power2-eq-square)
    qed

    have 2∗?s = a^2 − c^2
      using ⟨even (x + y)⟩ ⟨x∗y = a∗a⟩ ⟨z∗t = c∗c⟩ eq(1)
      by (simp add: power2-eq-square)

    have ?s > 0
      using ⟨x > 0⟩ ⟨y > 0⟩
      by auto

    have ?b ≥ 0 ?d ≥ 0
      by simp-all

    show ?thesis
```

```
    proof (cases ?b ≥ c)
      case True
      then show False
        using IMO2018SL-N5-lemma[of ?s a ?b c ?d]
        using ⟨?s^2 = a^2 + ?b^2⟩ ⟨?s^2 = c^2 + ?d^2⟩ ⟨2*?s = a^2 - c^2⟩
        using ⟨a > 0⟩ ⟨c > 0⟩ ⟨?s > 0⟩ ⟨?d ≥ 0⟩
        by simp
    next
      case False
      then have c ≥ ?b
        by simp
      then show False
        using IMO2018SL-N5-lemma[of ?s ?d c ?b a]
        using ⟨?s^2 = a^2 + ?b^2⟩ ⟨?s^2 = c^2 + ?d^2⟩ ⟨2*?s = a^2 - c^2⟩
        using ⟨a > 0⟩ ⟨c > 0⟩ ⟨?s > 0⟩ ⟨?b ≥ 0⟩ ⟨?d ≥ 0⟩
        by simp
    qed
  qed
qed

end
```