The mathcomp-eulerian formalization

A Stanley-style reading of permutation statistics in Rocq/MathComp

Contents

About this document

This document is a Stanley-style reading of the `mathcomp-eulerian` formalization. Each definition and statement is given twice: once in ordinary mathematical English (in the spirit of Stanley *Enumerative Combinatorics I*, Chapter 1) and once as the verbatim Rocq text, with a clickable link to the line in the repository on GitHub. The intention is that a mathematician can read top to bottom like a textbook and at any point click through to verify the formal statement.

Status. This edition covers all of Stanley §1.3 (cycles, Stirling numbers, inversions, major index, Foata), §1.4 (descents, Eulerian numbers, $\beta_n(S)$, q-analogues), §1.6.2 (longest alternating subsequence), and §1.6.3 (toggle action, ω-map, and the headline Corollary 1.6.5). Stanley §1.6.4 (André's reflection method) is included as Chapter 9 but is *partial*: the headline recurrence is proven modulo one named admit, in a file out of the active build chain. See that chapter for details and pointers to the retrospective document.

First-time readers. Read Chapter 1 once, then Chapters 2–9 top to bottom. Use Appendix B as a quick reference on second reading. The primer and glossary together contain everything a Stanley reader needs to know about mathcomp to verify the formal statements; you do not need to install Rocq.

Auxiliary lemmas. The narrative chapters cover the ∼70 most-cited results in the formalization. The full library contains ∼830 named results (Definition/Lemma/Theorem/Corollary), all of which are listed in Appendix A with their `coqdoc` docstrings and clickable links to GitHub. Auxiliary lemmas (those not narrated in a chapter) are flagged [aux]; narrated entries carry a [Ch X §$X.Y$] badge pointing back to where they appear in the body. The catalog is auto-generated from the `.v` files; adding a docstring to a result automatically enriches both the catalog and the deployed coqdoc HTML.

Conventions. The single recurring shift is 0-indexing. Stanley writes $w \in \mathfrak{S}_n$ for permutations of $\{1, \ldots, n\}$ and $\mathrm{D}(w) \subseteq \{1, \ldots, n-1\}$ for the descent set; we use {`perm 'I_n`} for permutations of $\{0, \ldots, n-1\}$ and {`set 'I_n.-1`} for the descent set. Concretely:

$$\text{our } \{\texttt{perm 'I\_n.+1}\} \;=\; \text{Stanley's } \mathfrak{S}_{n+1}.$$

A complete translation table is in `docs/READING_GUIDE.md`; a worked example through Stanley's $w = [3, 1, 4, 2]$ is in the same file.

Reading the formal text. Each Rocq block looks like this:

[Rocq, `descent.v:25`]

```
Definition descent_set s : {set 'I_n} := [set i | is_descent s i].
```

Decoded. $\mathrm{D}(s) = \{\, i \in \{0, \ldots, n-1\} : s(i) > s(i+1) \,\}$.

The bracketed link at the top is clickable and opens the exact line on GitHub. The Rocq body is verbatim; identifiers and keywords are highlighted but no mathematical substitution has been performed. The shaded *Decoded* line below the block restates the formal text in plain mathematics, mechanically substituting mathcomp casts for their meaning. Side by side, the two should say the same thing; if they do not, that is exactly the kind of discrepancy worth flagging. Some definitions also carry a Design choice sidebar that explains why a particular formalization was chosen when a more natural-looking alternative exists. These boxes are for first reading and can be skipped on a re-read.

What is *not* included here. Full proofs are not included; only the formal *statements* are shown alongside the informal mathematics, with a one-paragraph proof sketch for each non-trivial result. The full proof is always one click away in the linked source. The cd-index development (`mmtree.v`, `psi_*.v`) is project-internal scaffolding not in Stanley and is not covered.

Verifying axiom-freeness. Every theorem in this document is kernel-validated by Rocq with no axioms or admits. To check this yourself for, e.g., the headline Corollary 8.7:

```
echo 'From mathcomp_eulerian Require Import beta_swap.
      Print Assumptions beta_alt_max.' | rocq top -R . mathcomp_eulerian
```

prints `Closed under the global context`, meaning no axiom is used anywhere in its transitive dependencies.

Chapter 1

A combinatorialist's mathcomp primer

This chapter is a focused reference for the few mathcomp primitives that appear in Chapter 5. It is not a tutorial; it gives just enough vocabulary to read the formal blocks and recognise that they say what the informal text claims. Skim it once on first read; come back to it as a glossary when re-reading.

1.1 Ordinals: the type `'I_n`

The type `'I_n` (read "I-sub-n") is mathcomp's *finite type* of natural numbers strictly less than n. An inhabitant is a pair: a natural number `i : nat` and a *proof* `i < n`. The two together are written `Ordinal i (proof : i < n)` but a reader normally never constructs ordinals by hand — the type appears as the index of a sum or a permutation domain, and inhabitants come from the typing context.

`ord0 : 'I_n.+1` the value 0, with its proof $0 < n + 1$. Always available when the bound is at least 1.

`ord_max : 'I_n.+1` the value n, with its proof. The largest element of `'I_n.+1`.

`val i : nat` strips an ordinal back to its underlying natural number, forgetting the proof.

The *cardinality* of `'I_n` is exactly n (lemma `card_ord` in `fintype.v`). So `#|'I_4| = 4`.

1.2 Casts: `widen_ord`, `lift`, `unlift`

There are exactly two natural maps $'\mathtt{I}_n \to '\mathtt{I}_{n+1}$:

`widen_ord (h : n \leq n+1) i` sends `i : 'I_n` to the same numerical value, viewed inside the bigger type. Diagrammatically:

$$i : '\mathtt{I}_n \quad \xrightarrow{\texttt{widen\_ord}} \quad i : '\mathtt{I}_{n+1}.$$

The h argument is a proof that $n \leq n + 1$, supplied by `leqnSn n`. Mathcomp users write `widen_ord (leqnSn n) i` reflexively.

`lift (p : 'I_n.+1) i` sends `i : 'I_n` to a value in `'I_n.+1` *skipping* the value p. With $p =$ `ord0`, this shifts every value up by one:

$$i : 'I_n \quad \xrightarrow{\texttt{lift ord0}} \quad i+1 : 'I_{n+1}.$$

The general `lift p i` skips p, which is exactly what one needs when removing one position from a permutation.

The two casts together name two specific elements of `'I_n.+1` from the same source $i : 'I_n$: the value i itself, and the value $i+1$. This is the idiom that lets us write "the comparison between consecutive positions i and $i+1$" without ever leaving the well-typed world of finite ordinals.

The partial inverse of `lift p` is `unlift p`, of type `'I_n.+1 \to option 'I_n`. Concretely: `unlift p i = None` if `i = p`, and `unlift p i = Some j` otherwise, with j the unique witness of `i = lift p j`. This is the canonical Rocq idiom for the case-split "is the input the special index p, or one of the others?" — it appears in every inductive bijection on ordinals.

1.3 Finite sets: `{set T}`

For `T` a finite type (such as `'I_n`), `{set T}` is the type of *finite subsets of* T, with *extensional* equality: two sets are equal iff they have the same elements.

`[set i : T | P i]` set comprehension — the set of `i : T` satisfying the boolean predicate P. The vertical bar reads as "such that", as in mathematical notation.

`#|S|` the cardinality of `S` as a natural number.

`i \in S` membership test, returning a `bool`.

`~: S` complement, `S :|: T` union, `S :&: T` intersection, `S :\: T` difference.

`{set T}` is itself a finite type, which means big operators (sums, products) can be indexed by sets. This matters for the Eulerian recurrence and for the partition-by-descent-set identity `beta_eulerian`.

1.4 Permutations: `{perm T}`

`{perm T}` is the type of *bijections* `T \to T` for `T` a finite type. An inhabitant is a function together with a proof of bijectivity, packaged so that:

`s i` applies the permutation `s` to `i : T` as a function. (No special syntax is needed: `{perm T}` *coerces* to a function.)

`1%g` is the identity permutation. The `%g` selects mathcomp's group scope.

`(s * t)%g i` is composition. Mathcomp's convention is that `(s * t) i = t (s i)` (right-to-left for the *action*; the lemma is `permM`).

`perm (h : injective f)` constructs a permutation from an injective function `f : T \to T` (on a finite type, injective implies bijective).

The cardinality of `{perm 'I_n}` is $n!$ (Stanley's $|\mathfrak{S}_n|$); this is the lemma `card_perm` or `card_Sn` in mathcomp.

1.5 Big operators: \sum, \prod

Mathcomp's `bigop.v` provides indexed sums and products with a uniform syntax:

`\sum_(i : T) f i` sums `f i` over every `i : T` (`T` a finite type).

`\sum_(i < n) f i` same, with `i : 'I_n`.

`\sum_(i \in S) f i` sums over an explicit set `S`.

`\sum_(i | P i) f i` sums over indices satisfying the boolean filter `P`.

The vertical bar in `\sum_(i | P i)` is a *filter*, not the disjunction symbol. Replace `\sum` by `\prod` for products. Several recurring *moves* appear in the proofs:

`partition_big f xpredT` rewrite `\sum_x g x` as `\sum_y \sum_(x | f x = y) g x` — partition by the value of f.

`bigID p` split a sum into the part where the predicate p holds and the part where it does not.

`reindex ...` change of summation variable along a bijection.

These names appear in the proofs we will not reprint in this document, but they are worth recognising if you click through to a `.v` file.

1.6 Sections and section variables

Every formalization file in this project opens with

```
Section X.
Variable n : nat.
Definition foo := ...     (* implicitly takes n *)
Lemma bar : ...           (* implicitly takes n *)
End X.
```

Inside the section, definitions and lemmas may use `n` as if it were globally fixed. At `End X`, every definition and lemma is automatically generalised to take `n` as an explicit argument. This is why `descent.v` reads `Variable n : nat` once at the top and then `Definition is_descent s i := ...` without an explicit `n`, but outside the file `is_descent` is invoked as `is_descent n s i`. (For the mathematician: think of `Variable` as "Let `n` be a natural number" at the start of a chapter.)

1.7 The takeaway

Three points are enough to read Chapter 5:

1. `{perm 'I_n.+1}` is mathcomp's $\mathfrak{S}_{n+1}$. Apply a permutation to a position by writing `s i`; the descent set is a `{set 'I_n}` (positions, not values).
2. The two casts `widen_ord (leqnSn n) i` and `lift ord0 i` pick out positions i and $i+1$ inside `'I_n.+1` from a single $i : \mathtt{'I_n}$. The reader does not need to verify cast arithmetic by hand — the kernel rejects the file unless every cast is the unique one of its type.
3. Big operators `\sum_(...)` are just sums; the syntax variants control the index range. The vertical bar is a filter.

With these in hand, the formal blocks in Chapter 5 read directly. Each formal block is followed by a one-line *Decoded* reading that mechanically substitutes the mathcomp casts for their mathematical meaning, so the reader has both forms side by side.

Chapter 2

Cycles and Stirling numbers

This chapter formalizes Stanley §1.3.1 (cycle representation of permutations) and §1.3.2 (Stirling numbers of the first kind). The two main objects are the cycle count $c(w)$ and the (signless) Stirling cycle number $c(n, k)$ that counts permutations by $c(w)$. They are the cycle-side analogues of $\mathrm{des}(w)$ and $A(n, k)$ from Chapter 5.

2.1 Cycle count

Definition 2.1 (Cycle decomposition, cycle count)**.** Every $w \in \mathfrak{S}_n$ admits a unique decomposition into a set of *disjoint cycles*; let $c(w)$ denote the number of such cycles.

[Rocq, cycles.v:24]

```
Definition cycle_count {T : finType} (s : {perm T}) : nat :=
  #|porbits s|.
```

Decoded. $c(s)$ = number of orbits of the action of the cyclic group $\langle s \rangle$ on the underlying set — that is, the number of cycles in the disjoint-cycle decomposition.

Design choice: why `porbits`, not a manual cycle decomposition?

A naive formalization would build the cycle decomposition by hand: pick an element, follow the orbit, mark seen, repeat. Mathcomp's `porbits` (in `perm.v`) does this once and exposes the result as a `{set {set T}}` — a set of orbits — together with all the lemmas one would want (`porbits_partition`, `cover_porbits`, etc.). Reusing it costs nothing and gives proofs like `cycle_count_id` in three lines instead of thirty. The cardinality of the orbit set is exactly the cycle count.

Lemma 2.2 (Cycle count bound)**.** $c(w) \leq |T|$ *for every* $w \in \mathfrak{S}_T$*, with equality iff* w *is the identity.*

[Rocq, cycles.v:44]

```
Lemma cycle_count_le {T : finType} (s : {perm T}) :
  cycle_count s ≤ #|T|.
```

Decoded. The number of cycles is at most the number of elements (each cycle has at least one element).

[Rocq, cycles.v:64]

```
Lemma cycle_count_id (T : finType) :
  cycle_count (1 : {perm T}) = #|T|.
```

Decoded. The identity has every element as a fixed point, i.e. a 1-cycle; the cycle count equals $|T|$.

2.2 Stirling cycle numbers

Definition 2.3 (Signless Stirling cycle numbers)**.** The number $c(n, k)$ is the count of permutations of $[n] = \{1, \ldots, n\}$ with exactly k cycles.

[Rocq, cycles.v:33]

```
Definition stirling_c (n k : nat) : nat :=
  #|[set s : {perm 'I_n} | cycle_count s == k]|.
```

Decoded. $c(n, k) = |\{s \in \mathfrak{S}_n : c(s) = k\}|$, the number of perms of length n with k cycles. *Direct match to Stanley* (no off-by-one): we use `{perm 'I_n}`, of size n, exactly mirroring Stanley's S_n.

Lemma 2.4 (Row sum recovers $n!$)**.** $\sum_{k=0}^{n} c(n, k) = n!$

[Rocq, cycles.v:98]

```
Lemma stirling_c_row_sum_fact n :
  \sum_(k < n.+1) stirling_c n k = n`!.
```

Decoded. Sum over $k \in \{0, \ldots, n\}$ of $c(n, k)$ equals $n!$, since the sets partition $\mathfrak{S}_n$ by cycle count.

Proof sketch. The sets $\{s : c(s) = k\}$ for k ranging over all possible cycle counts partition $\mathfrak{S}_n$. Total cardinality is $|\mathfrak{S}_n| = n!$. □

Theorem 2.5 (Stirling cycle recurrence)**.** *For $n, k \geq 0$,*

$$c(n+1, k+1) \;=\; n \cdot c(n, k+1) \;+\; c(n, k).$$

[Rocq, stirling_fiber.v:341]

```
Lemma stirling_c_rec n k :
  stirling_c n.+1 k.+1 = n * stirling_c n k.+1 + stirling_c n k.
```

Decoded. Stanley's recurrence for the signless Stirling cycle numbers: a permutation of $[n+1]$ with $k+1$ cycles either inserts the element $n+1$ into one of the existing n cycle-positions of a perm of $[n]$ with $k+1$ cycles (n choices), or sits as a new fixed point in a perm of $[n]$ with k cycles.

Proof sketch. Partition perms of $[n+1]$ with $k+1$ cycles by whether $n+1$ is a fixed point. If yes, the rest is a perm of $[n]$ with k cycles (giving $c(n, k)$). If no, $n+1$ sits inside an existing cycle; there are n positions to insert it, giving $n \cdot c(n, k+1)$.

The formal proof uses an auxiliary construction `insert_after` (in `stirling_fiber.v`) that realises the bijection $[n] \times \{s \in \mathfrak{S}_n : c(s) = k+1\} \to \{\sigma \in \mathfrak{S}_{n+1} : \sigma(n+1) \neq n+1,\ c(\sigma) = k+1\}$. □

Chapter 3

Inversions and the major index

This chapter formalizes Stanley §1.3.3 (the inversion count inv and the major index maj, the two classical Mahonian statistics) and the basic identities each satisfies. The headline result — their equidistribution over $\mathfrak{S}_n$ — requires the Foata bijection and is the subject of Chapter 4.

3.1 Inversions

Definition 3.1 (Inversion)**.** A pair (i, j) with $1 \le i < j \le n$ is an *inversion* of $w \in \mathfrak{S}_n$ if $w_i > w_j$.

[Rocq, inversions.v:26]

```
Definition is_inv s (i j : 'I_n.+1) : bool :=
  (i < j) && (s j < s i).
```

Decoded. (i, j) is an inversion of s iff $i < j$ and $s(j) < s(i)$ (i.e. $s(i) > s(j)$).

Definition 3.2 (Inversion set, inversion count)**.** $\mathrm{Inv}(w) = \{(i, j) : i < j,\ w_i > w_j\}$, and $\mathrm{inv}(w) = |\mathrm{Inv}(w)|$.

[Rocq, inversions.v:30]

```
Definition inv_set s : {set 'I_n.+1 * 'I_n.+1} :=
  [set ij | is_inv s ij.1 ij.2].
```

Decoded. The set of inversion pairs.

[Rocq, inversions.v:34]

```
Definition inv s : nat := #|inv_set s|.
```

Decoded. $\mathrm{inv}(s)$ is the cardinality of the inversion set.

Lemma 3.3 (Identity has no inversions)**.** $\mathrm{inv}(\mathrm{id}) = 0$.

[Rocq, inversions.v:41]

```
Lemma inv_id : inv (1 : {perm 'I_n.+1}) = 0.
```

Decoded. The identity permutation has no (i, j) with $s(i) > s(j)$, hence no inversions.

Lemma 3.4 (Inversion bound)**.** $\mathrm{inv}(w) \le \binom{n}{2}$ *for every* $w \in \mathfrak{S}_n$.

[Rocq, inversions.v:136]

```
Lemma inv_le s : inv s ≤ 'C(n.+1, 2).
```

Decoded. $\mathrm{inv}(s) \le \binom{n+1}{2}$, the number of pairs (i, j) with $i < j$ in $\mathfrak{S}_{n+1}$.

Lemma 3.5 (Reversal identity for inv)**.** $\mathrm{inv}(w^{\mathrm{rev}}) = \binom{n}{2} - \mathrm{inv}(w)$.

[Rocq, inversions.v:219]

```
Lemma inv_rev_perm s :
  inv (rev_perm s) = 'C(n.+1, 2) - inv s.
```

Decoded. The reverse permutation has the "opposite" inversion count: each pair (i, j) with $i < j$ is either an inversion of s or of its reverse, but not both. Hence the two counts sum to $\binom{n+1}{2}$.

3.2 The major index

Definition 3.6 (Major index)**.** The *major index* of $w \in \mathfrak{S}_n$ is the sum of (1-indexed) descent positions:

$$\mathrm{maj}(w) \;=\; \sum_{i \in \mathrm{D}(w)} i.$$

[Rocq, inversions.v:81]

```
Definition maj s : nat := \sum_(i in descent_set s) (val i).+1.
```

Decoded. $\mathrm{maj}(s) = \sum_{i \in \mathrm{D}(s)} (i + 1)$. The `(val i).+1` adds 1 to recover Stanley's 1-indexing from our 0-indexed descent positions.

Design choice: why `(val i).+1` inside the sum, not in the descent set?

We could "correct" the indexing once for all by storing 1-indexed positions in the descent set. We don't, because the descent set type `{set 'I_n}` would no longer fit the ambient type system: 1-indexed positions in $\{1, \ldots, n-1\}$ would need `{set 'I_n.-1}` with values shifted by 1, and every set-level lemma (complement, partition, big-op) would carry a translation. Doing the index shift inside `maj` keeps the descent-set machinery clean and isolates the translation to the one definition that needs it. The price is a single `(val i).+1` in the formula; the benefit is that `descent_set`, `beta`, and `set_is_alt` all live in the same type without conversion casts.

Lemma 3.7 (Major-index bound)**.** $\mathrm{maj}(w) \le \binom{n}{2}$ *for every* $w \in \mathfrak{S}_n$.

[Rocq, inversions.v:158]

```
Lemma maj_le s : maj s ≤ 'C(n.+1, 2).
```

Decoded. $\mathrm{maj}(s) \le \binom{n+1}{2}$, attained by the reverse identity.

Example 3.8 (Stanley's $w = [3, 1, 4, 2]$, statistics)**.** Take $w = [3, 1, 4, 2] \in \mathfrak{S}_4$ as in Example 5.3. Pairs (i, j) with $w_i > w_j$: $(1, 2)$ since $3 > 1$, $(1, 4)$ since $3 > 2$, $(3, 4)$ since $4 > 2$. Hence $\mathrm{inv}(w) = 3$. Descent positions: $\mathrm{D}(w) = \{1, 3\}$, so $\mathrm{maj}(w) = 1 + 3 = 4$. In our types, $\mathrm{inv}(s) = 3$ and $\mathrm{maj}(s) = (0 + 1) + (2 + 1) = 4$, matching.

Chapter 4

Foata's bijection and MacMahon's equidistribution

This chapter formalizes Stanley §1.3.4: Foata's first fundamental bijection on $\mathfrak{S}_n$, which sends permutations of major index k bijectively to permutations of inversion count k. The corollary is MacMahon's equidistribution theorem.

4.1 The Foata transformation on words

The bijection is built from a recursive construction on words. Given a word u and a new letter a, classify the previous letters of u according to whether they are $< a$ or $> a$ (depending on the last letter of u), split u into blocks at the classifying letters, and cyclically rotate each block (last $\to$ front). Then append a.

[Rocq, foata.v:70-77]

```
Definition foata_step (a : nat) (u : seq nat) : seq nat :=
  match u with
  | [::] ⇒ [:: a]
  | _ :: _ ⇒
      let x := last 0 u in
      let P := if x < a then (fun y : nat ⇒ y < a) else (fun y ⇒ a
          < y) in
      flatten (map cyc_last_to_front (split_blocks P u)) ++ [:: a]
  end.
```

Decoded. Empty word: just $[a]$. Otherwise: pick the predicate P ("$< a$" or "$> a$" depending on the sign of last(u) vs a), `split_blocks P u` cuts u into blocks ending at P-letters, `cyc_last_to_front` rotates each block (last letter to front), `flatten` concatenates, then append a.

[Rocq, foata.v:80]

```
Definition foata (w : seq nat) : seq nat :=
  foldl (fun u a ⇒ foata_step a u) [::] w.
```

Decoded. Process w left-to-right, applying `foata_step` at each new letter starting from the empty word.

Example 4.1 (Stanley's running example)**.** $w = [3, 1, 4, 5, 9, 2, 6]$. Then:

$$\text{foata}(w) \;=\; [3, 4, 1, 5, 2, 9, 6].$$

Verified inside `foata.v` (lemma `sanity_inv_eq_maj` at line 125 confirms by `Compute`). Stanley's hand computation on p. 24 gives the same value.

4.2 Lifting to permutations

The seq-level bijection is wrapped into a permutation via `seq_to_perm`, which packages a uniqueness-and-bound proof obligation into a `{perm 'I_n.+1}`.

[Rocq, foata.v:1529]

```
Definition foata_perm (s : {perm 'I_n.+1}) : {perm 'I_n.+1} :=
  seq_to_perm (foata_perm_to_seq_size s) (foata_perm_to_seq_uniq s)
              (foata_perm_to_seq_bnd s).
```

Decoded. `foata_perm` s is the permutation whose one-line sequence is foata applied to the one-line sequence of s. The three named lemmas discharge the side-conditions (size, distinctness, bound) needed to package a `seq nat` back into a `{perm}`.

Design choice: why a seq-level bijection lifted to perms?

Foata's bijection is defined inductively on the word (seq), not on the permutation as a function. Trying to define it directly on `{perm 'I_n.+1}` would force every step to re-establish "the result is a permutation," an obligation that the seq-level construction discharges once at the end. We pay for the wrapping once (in `seq_to_perm`); in exchange the bijection itself has the natural, recursive form Stanley describes.

4.3 The equidistribution theorem

Theorem 4.2 ($\mathrm{inv}(\mathrm{foata}\, w) = \mathrm{maj}(w)$)**.** *For every* $w \in \mathfrak{S}_n$,

$$\mathrm{inv}(\mathrm{foata}(w)) \;=\; \mathrm{maj}(w).$$

[Rocq, foata.v:1576]

```
Lemma foata_perm_inv_maj n (s : {perm 'I_n.+1}) :
  inv (foata_perm s) = maj s.
```

Decoded. The Foata transformation converts maj-statistics to inv-statistics: for every s, the inversion count of foata(s) equals the major index of s.

Lemma 4.3 (`foata_perm` is injective)**.** *The map* $s \mapsto \mathrm{foata\_perm}\, s$ *is injective on* $\mathfrak{S}_{n+1}$.

[Rocq, foata.v:1586]

```
Lemma foata_perm_inj n : injective (@foata_perm n).
```

Decoded. Different inputs give different outputs (proven by constructing an explicit inverse `foata_inv` at the seq level).

Theorem 4.4 (MacMahon's equidistribution)**.** *For every* n, k*, the number of permutations of* $\mathfrak{S}_n$ *with* $\mathrm{inv} = k$ *equals the number with* $\mathrm{maj} = k$*:*

$$\#\{w \in \mathfrak{S}_n : \mathrm{inv}(w) = k\} \;=\; \#\{w \in \mathfrak{S}_n : \mathrm{maj}(w) = k\}.$$

[Rocq, foata.v:1598]

```
Theorem inv_maj_equidistr n k :
  #|[set s : {perm 'I_n.+1} | inv s == k]|
  = #|[set s : {perm 'I_n.+1} | maj s == k]|.
```

Decoded. The two count functions inv and maj have the same distribution over $\mathfrak{S}_{n+1}$.

Proof sketch. The map $s \mapsto \mathrm{foata\_perm}\, s$ sends $\{s : \mathrm{maj}(s) = k\}$ *into* $\{t : \mathrm{inv}(t) = k\}$ (Theorem 4.2); it is injective; on a finite set, injective implies surjective. So the two sets are in bijection, hence have equal cardinality. □

Chapter 5

Descents and Eulerian numbers

This chapter formalizes Stanley §1.4 (Stanley *Enumerative Combinatorics I*, 2nd ed., pp. 30–38). The central objects are the *descent set* of a permutation, the *Eulerian numbers* $A(n,k)$ that count permutations by descent count, and the set-refined count $\beta_n(S)$ that counts permutations by their full descent set.

5.1 The descent set

Definition 5.1 (Descent)**.** Let $w = (w_1, w_2, \ldots, w_n) \in \mathfrak{S}_n$. A position $i \in \{1, \ldots, n-1\}$ is a *descent* of w if $w_i > w_{i+1}$.

[Rocq, `descent.v:22`]

```
Definition is_descent s i : bool :=
  s (widen_ord (leqnSn n) i) > s (lift ord0 i).
```

Decoded. $s(i) > s(i+1)$ for $i \in \{0, \ldots, n-1\}$ and $s : \{0, \ldots, n\} \to \{0, \ldots, n\}$ a bijection.

Design choice: why `widen_ord` and `lift`, not `s i > s i.+1`?

The naive form `s i > s i.+1` does not type-check. The permutation `s : {perm 'I_n.+1}` expects an argument of type `'I_n.+1`, but a descent slot i lives in `'I_n` (there are n adjacencies for $n+1$ positions). The two casts `widen_ord (leqnSn n) i` and `lift ord0 i` are the canonical maps that ship i into the larger type `'I_n.+1` as the values i and $i+1$ respectively (Primer 1.2). There is no way to "just cast" — the two casts encode *which* element of `'I_n.+1` we mean. The verbosity is what makes the formalization correct by typing.

The result type `bool` (rather than `Prop`) makes `is_descent` computable: `Compute is_descent s i` returns `true` or `false` for any concrete `s` and `i`. Mathcomp's small-scale reflection bridges `is_descent s i = true` with the corresponding proposition on demand.

Definition 5.2 (Descent set)**.** The *descent set* of $w \in \mathfrak{S}_n$ is

$$\mathrm{D}(w) \;=\; \{\, i \in \{1, \ldots, n-1\} : w_i > w_{i+1} \,\}.$$

[Rocq, `descent.v:25`]

```
Definition descent_set s : {set 'I_n} := [set i | is_descent s i].
```

Decoded. $\mathrm{D}(s) = \{\, i \in \{0, \ldots, n-1\} : s(i) > s(i+1) \,\}$.

Design choice: why `{set 'I_n}`, not `pred 'I_n` or `seq 'I_n`?

Three options exist for representing a finite collection of positions: a boolean predicate (`pred T`), a list (`seq T`), or a set (`{set T}`). `{set 'I_n}` is the right choice here for two reasons. First, it has *extensional equality*: two sets are equal iff they have the same elements, so `descent_set s = D` is the condition we want, with no axiom of function extensionality required. Second, `{set 'I_n}` is itself a finite type, which means we can sum over it with mathcomp's big operators — the lemma `beta_eulerian` below uses `\sum_(D : {set 'I_n} | #|D| == k)` exactly because `{set}` supports this. `pred 'I_n` fails the first test (extensional equality of predicates needs `funext`); `seq 'I_n` carries spurious order and multiplicity.

$\mathrm{D}(w)$ is a subset of $\{1, \ldots, n-1\}$ in Stanley; in our types it is a subset of $\{0, \ldots, n-1\} = {}'\mathtt{I}_{\mathtt{n}}$. The correspondence is Stanley's $i \leftrightarrow$ our $(i-1)$.

Example 5.3 (Stanley's $w = [3,1,4,2]$, both ways)**.** Take $w = [3,1,4,2] \in \mathfrak{S}_4$. Then $\mathrm{D}(w) = \{1,3\}$ since $w_1 = 3 > 1 = w_2$ and $w_3 = 4 > 2 = w_4$. In our types, the corresponding $s : \{\mathtt{perm}\ {}'\mathtt{I}_4\}$ has $s\,0 = 2$, $s\,1 = 0$, $s\,2 = 3$, $s\,3 = 1$ (values shifted to $\{0,1,2,3\}$). Then:

```
Compute is_descent s 0.     (* = true, since s 0 = 2 > 0 = s 1 *)
Compute is_descent s 1.     (* = false, since s 1 = 0 < 3 = s 2 *)
Compute is_descent s 2.     (* = true, since s 2 = 3 > 1 = s 3 *)
Compute descent_set s.      (* = [set 0; 2] : {set 'I_3} *)
Compute des s.              (* = 2 *)
```

After the index shift our $\{0,2\}$ corresponds to Stanley's $\{1,3\}$, and $\mathrm{des}(s) = 2 = \mathrm{des}(w)$. The computation is what the kernel does on the elaborated term — the formal definition matches the informal one not by editorial promise but by the test running here on the page.

Definition 5.4 (Descent number, ascent number)**.** $\mathrm{des}(w) = |\mathrm{D}(w)|$ and $\mathrm{asc}(w) = (n-1) - \mathrm{des}(w)$.

[Rocq, descent.v:27]

```
Definition des s : nat := #|descent_set s|.
```

Decoded. $\mathrm{des}(s) = |\mathrm{D}(s)|$, the number of descent positions.

[Rocq, descent.v:29]

```
Definition asc s : nat := n - des s.
```

Decoded. $\mathrm{asc}(s) = n - \mathrm{des}(s)$, the number of ascent positions (here n is the number of descent slots, i.e. Stanley's $n-1$).

Design choice: why is the parameter `n` "perm size minus one"?

A permutation of $n+1$ elements has n adjacencies, so the descent set lives in $\{0, \ldots, n-1\}$ regardless of how we parameterise. Two choices: name the parameter after the perm size (forcing every descent-set type to be `{set 'I_n.-1}` with a `.-1`), or name it after the descent slot count (forcing the perm type to be `{perm 'I_n.+1}` with a `.+1`). We picked the latter because it removes a `.-1` from every theorem about descent sets and from every `eulerian n k` statement. The cost is a single conversion: **our parameter `n` is Stanley's** $n-1$, and our `eulerian n k` is Stanley's $A(n+1,k)$. This is a project convention, not a mathcomp requirement; mathcomp itself uses both styles in different files. We flag the off-by-one explicitly at every theorem statement that crosses it.

Lemma 5.5 (Descent count is bounded)**.** $\mathrm{des}(w) \le n-1$ *for every* $w \in \mathfrak{S}_n$.

[Rocq, descent.v:38]

```
Lemma des_le s : des s ≤ n.
```

Decoded. $\mathrm{des}(s) \le n$ for every $s : \{\mathtt{perm}\ {}'\mathtt{I}_{n+1}\}$ (equivalently $\mathrm{des}(w) \le |w| - 1$).

Proof sketch. $\mathrm{D}(w) \subseteq \{0, \ldots, n-1\}$ which has n elements; the cardinality is at most n. □

Lemma 5.6 (Identity has no descents)**.** $\mathrm{des}(\mathrm{id}) = 0$.

[Rocq, descent.v:44]

```
Lemma des_id : des (1 : {perm 'I_n.+1}) = 0.
```

Decoded. $\mathrm{des}(\mathrm{id}) = 0$, where `1%g` is the group identity, i.e. the identity permutation.

Proof sketch. For the identity, $s\,i = i$ at every position, so $s(i) = i < i+1 = s(i+1)$ — no descent slot. □

5.2 Reversal symmetry

A useful symmetry: reversing a permutation flips ascents and descents at corresponding positions.

Definition 5.7 (Reverse permutation)**.** Let $w \in \mathfrak{S}_n$ and define w^{rev} by $w^{\mathrm{rev}}_i = w_{n+1-i}$.

[Rocq, descent.v:80]

```
Definition rev_perm_ord : {perm 'I_n.+1} := perm (@rev_ord_inj
    n.+1).
```

Decoded. The position-reversal involution $i \mapsto n - i$, packaged as a permutation. `perm (h : injective f)` is the constructor that turns an injective function into a `{perm T}`.

[Rocq, descent.v:85]

```
Definition rev_perm s : {perm 'I_n.+1} := rev_perm_ord * s.
```

Decoded. $w^{\mathrm{rev}}(i) = s(n-i)$, the reversal of s viewed at the level of positions.

Design choice: why `rev_perm_ord * s`, not `s * rev_perm_ord`?

Mathcomp's group convention is `(s * t) i = t (s i)` (see the lemma `permM`). With `rev_perm := rev_perm_ord * s`, applying gives `(rev_perm_ord * s) i = s (rev_perm_ord i) = s (n - i)`. This reverses *positions* (Stanley's $w^{\mathrm{rev}}_i = w_{n+1-i}$). Post-composing instead would reverse *values*, a different operation. Picking the wrong side here would silently change the lemma's content; the choice matches Stanley by direct check on a small example.

Lemma 5.8 (Descents of the reverse)**.** *Position i is a descent of w^{rev} iff position $n-i$ is* not *a descent of w:*

$$i \in \mathrm{D}(w^{\mathrm{rev}}) \iff (n-i) \notin \mathrm{D}(w).$$

[Rocq, descent.v:90]

```
Lemma is_descent_rev s (i : 'I_n) :
  is_descent (rev_perm s) i = ~~ is_descent s (rev_ord i).
```

Decoded. Position i is a descent of w^{rev} iff position $n-i$ is *not* a descent of w. The `~~` is boolean negation.

Proof sketch. Direct calculation: $(\texttt{rev\_perm}\,\texttt{s})\,\texttt{j} = s(n-j)$, and the comparison $s(n-i) > s(n-i-1)$ is the negation of $s(n-i-1) > s(n-i)$. □

5.3 Eulerian numbers

Definition 5.9 (Eulerian number)**.** The *Eulerian number* $A(n,k)$ is the number of permutations of $\{1,\dots,n\}$ with exactly k descents:

$$A(n,k) \;=\; \#\{\, w \in \mathfrak{S}_n : \mathrm{des}(w) = k \,\}.$$

[Rocq, eulerian.v:14]

```
Definition eulerian (n k : nat) : nat :=
  #|[set s : {perm 'I_n.+1} | des s == k]|.
```

Decoded. `eulerian` $n\,k = |\{s \in \mathfrak{S}_{n+1} : \mathrm{des}(s) = k\}|$. The `==` is boolean equality on `nat`, and `#|...|` is set cardinality.

Off-by-one alert. The Rocq parameter n is "permutation size minus one" (see the design-choice sidebar above). Concretely, `eulerian n k` counts perms of $\{\texttt{perm 'I}_{n+1}\}$, which are Stanley's $\mathfrak{S}_{n+1}$. Hence

$$\texttt{eulerian}\ n\ k\ =\ A(n+1,k).$$

For instance, `eulerian 3 1` corresponds to Stanley's $A(4,1) = 11$ (Stanley table p. 36).

```
Compute eulerian 0 0.     (* = 1      -- Stanley A(1,0) = 1 *)
Compute eulerian 1 0.     (* = 1      -- Stanley A(2,0) = 1 *)
Compute eulerian 1 1.     (* = 1      -- Stanley A(2,1) = 1 *)
Compute eulerian 2 0.     (* = 1      -- Stanley A(3,0) = 1 *)
Compute eulerian 2 1.     (* = 4      -- Stanley A(3,1) = 4 *)
Compute eulerian 2 2.     (* = 1      -- Stanley A(3,2) = 1 *)
```

The numbers match Stanley's table on p. 36. Computing `eulerian` on `{perm 'I_n.+1}` requires enumerating the set; for $n \geq 4$ the brute-force enumeration is slow, but for the small cases above it returns immediately and the values agree.

Lemma 5.10 (Row sum)**.** $\sum_{k=0}^{n-1} A(n,k) = n!$

[Rocq, eulerian.v:31]

```
Lemma eulerian_row_sum_fact n :
  \sum_(k < n.+1) eulerian n k = n.+1`!.
```

Decoded. $\sum_{k=0}^{n} A(n+1,k) = (n+1)!$. The `` `! `` is mathcomp's factorial notation; `\sum_(k<n.+1)` sums over $k \in \{0,\ldots,n\}$.

Proof sketch. The summands partition $\mathfrak{S}_{n+1}$ by descent count; the total is $|\mathfrak{S}_{n+1}| = (n+1)!$. □

Lemma 5.11 (Out-of-range vanishing)**.** $A(n,k) = 0$ *for* $k \geq n$.

[Rocq, eulerian.v:34]

```
Lemma eulerian_out_of_range n k :
  n < k → eulerian n k = 0.
```

Decoded. If $k > n$ then $A(n+1,k) = 0$, since perms of length $n+1$ have at most n descents.

Proof sketch. A permutation has at most n descent positions (Lemma 5.5); none can have $k > n$ descents. □

Lemma 5.12 (The unique perm with no descents is the identity)**.** *If* $\mathrm{des}(w) = 0$ *then* $w = \mathrm{id}$.

[Rocq, eulerian.v:41]

```
Lemma des0_id n (s : {perm 'I_n.+1}) : des s = 0 → s = 1%g.
```

Decoded. If $\mathrm{des}(s) = 0$ then s is the identity permutation.

Proof sketch. $\mathrm{des}(s) = 0$ means $s(i) < s(i+1)$ for every adjacent pair i, so s is strictly increasing. The only strictly increasing function $\texttt{'I}_{n+1} \to \texttt{'I}_{n+1}$ is the identity. □

5.4 The insert-max bijection

A central tool for the Eulerian recurrence is the bijection that inserts the maximum value $n+1$ at a chosen position into a permutation of $\{1,\dots,n\}$.

Definition 5.13 (Insert-max)**.** For $t \in \mathfrak{S}_n$ and $p \in \{1,\dots,n+1\}$, let $\texttt{insert\_max}(t,p) \in \mathfrak{S}_{n+1}$ be the permutation obtained by inserting the value $n+1$ at position p in the one-line form of t.

[Rocq, eulerian.v:134-157]

```
Definition insert_max_fun (i : 'I_n.+2) : 'I_n.+2 :=
  match unlift p i with
  | Some j ⇒ widen_ord (leqnSn _) (t j)
  | None ⇒ ord_max
  end.

Definition insert_max_perm : {perm 'I_n.+2} := perm
    insert_max_fun_inj.
```

Decoded. For $i \in \{0,\dots,n+1\}$: if $i = p$, output $n+1$ (the new max); otherwise, write $i = \texttt{lift p j}$ for the unique $j \in \{0,\dots,n\}$ that skips p, and output $t(j)$ cast into the larger type.

Design choice: why `match unlift p i with`?

The fundamental case-split for inserting at position p is "is this position p or not?". Mathcomp encodes this as the `option`-valued partial inverse `unlift p`: it returns `None` exactly when $i = p$, and `Some j` otherwise — where $j : \texttt{'I}_{n+1}$ is the unique witness of $i = \texttt{lift p j}$. Carrying j in the `Some` branch (rather than recomputing it from i and p) avoids any arithmetic juggling: we get the correct `'I_n.+1` value with its bound proof for free, and can apply t directly. This is the universal idiom for constructing permutations on `'I_n.+2` from a permutation on `'I_n.+1` plus a special index.

Definition 5.14 (Extract-max)**.** The inverse: given $\sigma \in \mathfrak{S}_{n+1}$ with $\sigma(p) = n+1$ for some position p, remove that value and standardise to obtain $t \in \mathfrak{S}_n$.

[Rocq, eulerian.v:185-203]

```
Definition extract_max_fun (j : 'I_n.+1) : 'I_n.+1 :=
  odflt j (unlift ord_max (s (lift p j))).

Definition extract_max_perm : {perm 'I_n.+1} := perm
    extract_max_fun_inj.
```

Decoded. For $j \in \{0,\dots,n\}$: look at s at position j of the original (skipping p); the result is some value in $\{0,\dots,n+1\}$ that is not $n+1$ (because $\sigma(p) = n+1$ and σ is a bijection); strip the high bit to land in $\{0,\dots,n\}$. The `odflt j (...)` provides a default that is provably unreachable.

Theorem 5.15 (Insert-max is a bijection)**.** *The map* $(t,p) \mapsto \texttt{insert\_max}(t,p)$ *is a bijection* $\mathfrak{S}_n \times \{1,\dots,n+1\} \to \mathfrak{S}_{n+1}$.

[Rocq, eulerian.v:437]

```
Lemma insert_max_perm_bij n :
  ∀ sigma : {perm 'I_n.+2},
    ∃! tp : {perm 'I_n.+1} * 'I_n.+2,
      sigma = insert_max_perm tp.1 tp.2.
```

Decoded. Every $\sigma \in \mathfrak{S}_{n+2}$ arises uniquely as $\texttt{insert\_max}(t,p)$ for some $(t,p) \in \mathfrak{S}_{n+1} \times \{0,\dots,n+1\}$.

Proof sketch. Given σ, the unique p is $\sigma^{-1}(n+1)$ and $t = \texttt{extract\_max}(\sigma,p)$. The two roundtrips are direct calculations. □

5.5 Eulerian recurrence

Theorem 5.16 (Eulerian recurrence)**.** *For $n \geq 1$ and $0 \leq k \leq n$,*

$$A(n+1,k) \;=\; (k+1)\,A(n,k) \;+\; (n-k+1)\,A(n,k-1).$$

[Rocq, eulerian.v:458]

```
Theorem eulerian_rec n k :
  eulerian n.+1 k =
    (k.+1) * eulerian n k + (n.+1 - k) * eulerian n k.-1.
```

Decoded. $A(n+2,k) = (k+1)\,A(n+1,k) + (n+1-k)\,A(n+1,k-1)$. The `n.+1 - k` on the second term reads as "$n+1-k$" under our parameterisation; compare to Stanley's $(n+1-k)$ in $A(n+2,k) = (k+1)A(n+1,k) + (n+1-k)\,A(n+1,k-1)$.

Proof sketch. Using the insert-max bijection, classify each $\sigma \in \mathfrak{S}_{n+1}$ by the position p of $n+1$. There are $k+1$ ascent positions and $n-k$ descent positions where inserting $n+1$ preserves the descent count; inserting at the other $n-k+1$ slots increases it by one. Summing recovers the recurrence. □

5.6 The set-refined count β

The descent count is a coarse invariant. The full descent *set* carries more information; the count of permutations with a fixed descent set is a finer object.

Definition 5.17 (β count)**.** For a subset $S \subseteq \{1,\dots,n-1\}$, the count $\beta_n(S)$ is the number of permutations of $\{1,\dots,n\}$ whose descent set is exactly S:

$$\beta_n(S) \;=\; \#\{\, w \in \mathfrak{S}_n : \mathrm{D}(w) = S \,\}.$$

[Rocq, beta.v:19]

```
Definition beta (n : nat) (D : {set 'I_n}) : nat :=
  #|[set sigma : {perm 'I_n.+1} | descent_set sigma == D]|.
```

Decoded. `beta` $n\,D = |\{\sigma \in \mathfrak{S}_{n+1} : \mathrm{D}(\sigma) = D\}|$, the number of perms of length $n+1$ with descent set exactly D.

Off-by-one alert. As with `eulerian`, the Rocq parameter n is "perm size minus one." So `beta n D` for $D : \{\texttt{set 'I\_n}\}$ equals Stanley's $\beta_{n+1}(D)$, with D interpreted as a subset of $\{0,\dots,n-1\}$ that corresponds to Stanley's 1-indexed $S \subseteq \{1,\dots,n\}$.

Lemma 5.18 (β partitions Eulerian)**.** *Summing $\beta_n(S)$ over all S of cardinality k recovers the Eulerian number:*

$$\sum_{|S|=k} \beta_n(S) \;=\; A(n,k).$$

[Rocq, beta.v:124]

```
Lemma beta_eulerian n k :
  \sum_(D : {set 'I_n} | #|D| == k) beta D = eulerian n k.
```

Decoded. $\sum_{D \subseteq \{0,\dots,n-1\},\ |D|=k} \beta_{n+1}(D) = A(n+1,k)$. The big-op syntax `\sum_(D : {set 'I_n} | #|D| == k)` sums over all sets D of cardinality exactly k.

Proof sketch. The sets $\{\sigma : \mathrm{D}(\sigma) = D\}$ for D ranging over subsets of cardinality k partition the set of permutations with k descents. The cardinalities sum. □

Lemma 5.19 (β is reversal-invariant)**.** *For $S \subseteq \{1, \ldots, n-1\}$, let $S^{\mathrm{rev}} = \{n - i : i \in S\}$. Then $\beta_n(S^{\mathrm{rev}}) = \beta_n(S^{\mathrm{c}})$, where S^{c} is the complement in $\{1, \ldots, n-1\}$.*

[Rocq, beta.v:140]

```
Lemma beta_rev n (D : {set 'I_n}) :
  beta [set rev_ord i | i in D] = beta (~: D).
```

Decoded. $\beta_{n+1}(\{n-1-i : i \in D\}) = \beta_{n+1}(\overline{D})$. The set comprehension `[set rev_ord i | i in D]` is the image of D under $i \mapsto n-1-i$; `~: D` is the complement of D in `'I_n`.

Proof sketch. Reversal $w \mapsto w^{\mathrm{rev}}$ is a bijection on $\mathfrak{S}_n$; by Lemma 5.8 it maps descent set D to $\{n-i : i \notin D\}$, which is the complement of $\{n-i : i \in D\}$. Counting on both sides matches. □

5.7 What this chapter doesn't cover

This chapter establishes the basic objects (D, des, A, β) and the foundational identities. The headline of the formalization,

$$\beta_n(S) \leq E_n \quad \text{with equality iff } S \text{ is alternating} \quad \text{(Stanley Cor 1.6.5)}$$

requires the toggle-action machinery of §1.6.3 and is the subject of a future chapter.

Chapter 6

The q-analogues

This chapter formalizes the q-extensions of §1.4: the q-factorial $[n]_q!$ as the inversion generating function over $\mathfrak{S}_n$, and the bivariate q-Eulerian polynomial $A_n(q,t)$ as the joint generating function for $(\mathrm{maj}, \mathrm{des})$. The two specializations $t = 1$ and $q = 1$ recover $[n]_q!$ and the classical Eulerian polynomial respectively.

6.1 q-integers and the q-factorial

Definition 6.1 (q-integer)**.** $[n]_q = 1 + q + q^2 + \cdots + q^{n-1}$.

[Rocq, qfact.v:25]

```
Definition q_int (n : nat) : {poly int} := \sum_(i < n) 'X^i.
```

Decoded. $[n]_q$ as a polynomial in the formal indeterminate `'X`, with integer coefficients. `\sum_(i<n) 'X^i` sums X^i for $i \in \{0, \ldots, n-1\}$.

Definition 6.2 (q-factorial)**.** $[n]_q! = [1]_q \cdot [2]_q \cdots [n]_q$.

[Rocq, qfact.v:26]

```
Definition q_fact (n : nat) : {poly int} := \prod_(k < n.+1) q_int
   k.+1.
```

Decoded. `q_fact` $n = [1]_q \cdot [2]_q \cdots [n+1]_q = [n+1]_q!$. The off-by-one is the same as in `eulerian`: our parameter is "size minus one," so `q_fact` $n = [n+1]_q!$ in Stanley's notation.

Theorem 6.3 (inv-generating function for $\mathfrak{S}_n$)**.** $\sum_{w \in \mathfrak{S}_n} q^{\mathrm{inv}(w)} = [n]_q!$

[Rocq, qfact.v:188]

```
Theorem inv_q_fact n :
  \sum_(σ : {perm 'I_n.+1}) 'X^(inv σ) = q_fact n :> {poly int}.
```

Decoded. The polynomial $\sum_{\sigma \in \mathfrak{S}_{n+1}} X^{\mathrm{inv}(\sigma)}$ is exactly $[n+1]_q!$.

Proof sketch. Induction on n: at each step, classify permutations of $\mathfrak{S}_{n+2}$ by where the maximum was inserted (using `insert_max_perm_bij`); inserting at position p adds exactly $n + 1 - p$ inversions, so the sum factors as $[n+1]_q! \cdot [n+2]_q$. □

Theorem 6.4 (maj-generating function via Foata)**.** $\sum_{w \in \mathfrak{S}_n} q^{\mathrm{maj}(w)} = [n]_q!$

[Rocq, qfact.v:218]

```
Theorem maj_q_fact n :
  \sum_(σ : {perm 'I_n.+1}) 'X^(maj σ) = q_fact n :> {poly int}.
```

Decoded. Same identity as `inv_q_fact` but for maj instead of inv — a direct corollary of MacMahon's equidistribution.

Proof sketch. Reindex the maj-sum along Foata's bijection: $\sum_\sigma X^{\mathrm{maj}(\sigma)} = \sum_\sigma X^{\mathrm{inv}(\mathrm{foata}(\sigma))}$ which equals $[n+1]_q!$ by the previous theorem. □

6.2 The bivariate q-Eulerian polynomial

Definition 6.5 (q-Eulerian polynomial)**.** $A_n(q,t) = \sum_{w \in \mathfrak{S}_n} q^{\mathrm{maj}(w)} \cdot t^{\mathrm{des}(w)}$.

[Rocq, qeul.v:37]

```
Definition q_eul_pol (n : nat) : {poly {poly int}} :=
  \sum_(σ : {perm 'I_n.+1}) qpow (maj σ) * 'X^(des σ).
```

Decoded. $A_{n+1}(q,t) = \sum_{\sigma \in \mathfrak{S}_{n+1}} q^{\mathrm{maj}(\sigma)} t^{\mathrm{des}(\sigma)}$. Bivariate: outer indeterminate `'X` is t, inner (under `qpow`) is q.

Design choice: why bivariate `{poly {poly int}}`?

The natural type for "polynomial in t with coefficients polynomials in q" is the iterated polynomial ring `{poly {poly int}}`. Mathcomp's polynomial library is fully composable: `{poly R}` is itself a ring (in fact a commutative ring if R is), so `{poly {poly int}}` inherits all the algebra. The trade-off is that the syntax is heavier — the constant-poly embedding `_%:P`, the notation `qpow k` (for the inner q^k embedded as an outer constant), and `horner_eval` for substitution all need to be juggled. We name them in advance to keep the recurrence readable.

Theorem 6.6 (Specialize $t := 1$)**.** $A_n(q,1) = [n]_q!$

[Rocq, qeul.v:44]

```
Theorem q_eul_pol_t1 n :
  (q_eul_pol n).[1] = q_fact n.
```

Decoded. Evaluating $A_{n+1}(q,t)$ at $t = 1$ gives $\sum_\sigma q^{\mathrm{maj}(\sigma)} \cdot 1^{\mathrm{des}(\sigma)} = \sum_\sigma q^{\mathrm{maj}(\sigma)} = [n+1]_q!$ (the previous theorem).

Definition 6.7 (Classical Eulerian polynomial)**.** $A_n(t) = \sum_{k \geq 0} A(n,k) \cdot t^k$.

[Rocq, qeul.v:58]

```
Definition eul_pol (n : nat) : {poly int} :=
  \sum_(k < n.+1) (eulerian n k)%:R * 'X^k.
```

Decoded. `eul_pol` $n = \sum_{k=0}^{n} A(n+1,k) \cdot t^k$ (the off-by-one inherited from `eulerian`).

Theorem 6.8 (Specialize $q := 1$)**.** $A_n(1,t) = A_n(t)$, *the classical Eulerian polynomial.*

[Rocq, qeul.v:84]

```
Theorem q_eul_pol_q1 n :
  q1_subst (q_eul_pol n) = eul_pol n.
```

Decoded. Substituting $q = 1$ in each coefficient of $A_{n+1}(q,t)$ recovers the classical Eulerian polynomial. The substitution is implemented by `q1_subst` = `map_poly (horner_eval 1)`, which evaluates each inner q-polynomial at $q = 1$.

Chapter 7

Longest alternating subsequence

This chapter formalizes Stanley §1.6.2: the longest alternating subsequence statistic $\mathrm{as}(w)$, with the closed-form characterization

$$\mathrm{as}(w) \;=\; \mathrm{turn\_count}(w) + 2,$$

where the right-hand side is two more than the number of "turning points" (positions where the direction of w changes).

7.1 Turning points

Definition 7.1 (Turning point)**.** Position i is a *turning point* of w if the direction at position i differs from the direction at position $i+1$: exactly one of $\{i, i+1\}$ is a descent position.

[Rocq, altsub.v:41]

```
Definition is_turn s (i : 'I_n) : bool :=
  is_descent s (widen_ord (leqnSn _) i) (+) is_descent s (lift ord0
     i).
```

Decoded. is_turn $s\,i$ is the boolean XOR of two adjacent `is_descent` indicators. For $s : \{\texttt{perm}\ 'I_{n+2}\}$ and $i : 'I_n$, the two arguments of XOR are descent slots i and $i+1$ (both in $'I_{n+1}$, via the same widen+lift idiom as Chapter 5).

[Rocq, altsub.v:44]

```
Definition turn_count s : nat :=
  #|[set i : 'I_n | is_turn s i]|.
```

Decoded. turn_count(s) = number of turning positions.

7.2 Alternating subsequences

Definition 7.2 (Alternating sequence of naturals)**.** A seq $x_1, x_2, \ldots, x_k$ is *alternating* if $x_1 < x_2 > x_3 < \cdots$ or $x_1 > x_2 < x_3 > \cdots$.

[Rocq, altsub.v:71]

```
Definition is_alt (xs : seq nat) : bool :=
  match xs with
  | [::] ⇒ true
  | [:: _] ⇒ true
  | x :: y :: xs' ⇒
```

```
      ((x < y) && alt_aux false y xs') || ((y < x) && alt_aux true
         y xs')
  end.
```

Decoded. Empty and singleton seqs are alternating. Otherwise we try both starting directions: "up first" (`x < y` then `alt_aux false`) or "down first" (`y < x` then `alt_aux true`); is_alt holds iff either branch succeeds. The auxiliary `alt_aux` alternates the boolean parity at each step.

Definition 7.3 (Permutation as a seq, subseq at chosen positions)**.**

[Rocq, altsub.v:84]

```
Definition perm_seq n (s : {perm 'I_n.+2}) : seq nat :=
  [seq val (s i) | i <- enum 'I_n.+2].
```

Decoded. The one-line form of s: the seq $[s(0); s(1); \ldots; s(n+1)]$ of natural numbers.

[Rocq, altsub.v:97]

```
Definition pick_seq n (s : {perm 'I_n.+2}) (I : {set 'I_n.+2}) :
   seq nat :=
  [seq val (s j) | j <- sort (fun a b ⇒ val a ≤ val b) (enum I)].
```

Decoded. For an index set $I \subseteq \{0, \ldots, n+1\}$, sort the positions in I in increasing order and pick the corresponding values from s. This is "the subsequence of s at positions I, in order."

7.3 The two definitions of as

Definition 7.4 (Bijective form: as_perm_max)**.** as_perm_max$(s) = \max\{|I| : \text{pick_seq}(s, I) \text{ is alternating}\}$.

[Rocq, altsub.v:105]

```
Definition as_perm_max n (s : {perm 'I_n.+2}) : nat :=
  \max_(I : {set 'I_n.+2} | is_alt (pick_seq s I)) #|I|.
```

Decoded. The maximum cardinality over all index sets I whose sorted-value picked subseq is alternating. This is the geometric "length of the longest alternating subseq" definition.

Definition 7.5 (Closed-form: as_perm)**.** as_perm$(s) =$ turn_count$(s) + 2$.

[Rocq, altsub.v:109]

```
Definition as_perm n (s : {perm 'I_n.+2}) : nat := (turn_count
   s).+2.
```

Decoded. Two more than the number of turning positions. This is Stanley's *closed-form* characterization, taken here as a definition; the equivalence with the bijective form is the headline theorem below.

7.4 The headline equivalence

Theorem 7.6 (Stanley's characterization, §1.6.2)**.** *For every* $s \in \mathfrak{S}_{n+2}$, as_perm_max$(s) =$ turn_count$(s) + 2$.

[Rocq, altsub.v:2114]

```
Theorem as_perm_max_eq s : as_perm_max s = (turn_count s).+2.
```

Decoded. The longest alternating subsequence of s has length exactly turn_count$(s) + 2$.

Design choice: why this took five sessions

The proof has two halves — $\geq$ (existence of a long alt subseq) and $\leq$ (no longer one exists). The lower bound is constructive: pick the boundary endpoints plus one position per turning point. The upper bound is where the difficulty was: the naive "leftmost turn in $[\sigma_i, \sigma_{i+2})$" injection from sign-flips of the picked subseq to turning positions of s *is not injective* — adjacent sign-flips can share a single turning window. The proof in `altsub.v` pivots to a global bound via the seq-level statistic `flip_count`, which is monotone under taking subseqs: `flip_count(sign_seq(pick_seq` $s\, I$`))` $\leq$ `flip_count(sign_seq(perm_seq` s`))` $=$ `turn_count`(s). The detour is recorded in `docs/plans/ALTSUB_PLAN.md`.

Chapter 8

Toggle action and Stanley's Corollary 1.6.5

This chapter formalizes Stanley §1.6.3: the toggle action on descent sets, the ω-map, and the headline corollary that the alternating descent set strictly maximizes $\beta_n(S)$ — the central theorem of this project.

8.1 Symmetric difference and single-position toggle

Definition 8.1 (Symmetric difference, single-position toggle)**.**

[Rocq, beta_omega.v:25]

```
Definition sym_diff (n : nat) (D E : {set 'I_n}) : {set 'I_n} :=
  (D :\: E) :|: (E :\: D).
```

Decoded. Standard set-theoretic symmetric difference $D \triangle E$.

[Rocq, beta_omega.v:28]

```
Definition toggle_at (n : nat) (D : {set 'I_n}) (i : 'I_n) : {set
    'I_n} :=
  sym_diff D [set i].
```

Decoded. toggle_at$(D, i) = D \triangle \{i\}$. Adds i if absent, removes it if present.

8.2 The ω-map

For a descent set $D \subseteq \{1, \ldots, n-1\}$ in Stanley's $\mathfrak{S}_n$, $\omega(D)$ records the positions where adjacent membership in D changes.

[Rocq, beta_omega.v:57]

```
Definition omega_set (n : nat) (D : {set 'I_n.+1}) : {set 'I_n} :=
  [set k : 'I_n | (widen_ord (leqnSn n) k \in D) != (lift ord0 k
     \in D)].
```

Decoded. $k \in \omega(D)$ iff membership of position k in D *differs* from membership of position $k+1$ in D. The two membership tests use the now-familiar widen+lift idiom to view k and $k+1$ inside the larger type.

8.3 The alternating descent set

Definition 8.2 (Alternating descent pattern)**.** $D^n_{\text{alt}} = \{1, 3, 5, \dots\} \cap \{1, \dots, n-1\}$ (Stanley's 1-indexed odd positions).

[Rocq, beta_swap.v:31]

```
Definition alt_desc_set (n : nat) : {set 'I_n} :=
  [set i : 'I_n | ~~ odd i].
```

Decoded. The set of *even-indexed* positions $\{0, 2, 4, \dots\}$ of $'\mathtt{I}_n$. Under our 0-indexing, this matches Stanley's 1-indexed odd positions $\{1, 3, 5, \dots\}$.

[Rocq, beta_swap.v:40]

```
Definition set_is_alt (n : nat) (D : {set 'I_n}) : bool :=
  [∀ i : 'I_n, [∀ j : 'I_n,
     (val j == (val i).+1) ⟹ ((i \in D) != (j \in D))]].
```

Decoded. set_is_alt(D): every consecutive pair $(i, i+1)$ in $'\mathtt{I}_n$ has differing membership in D. Equivalently, D is either D^n_{alt} or its complement (lemma `set_is_alt_classify` below).

8.4 Value-complement and β symmetry

[Rocq, beta_swap.v:128]

```
Definition compl_perm n (s : {perm 'I_n.+1}) : {perm 'I_n.+1} :=
  s * rev_perm_ord n.
```

Decoded. The value-complement of s: compl_perm$(s)\,i = n - s(i)$. Defined by post-composition with the value-reversal involution.

Lemma 8.3 (Descent set under value-complement)**.** D(compl_perm$(s)) = \overline{\text{D}(s)}$ *(the complement in* $'\mathtt{I}_n$*).*

[Rocq, beta_swap.v:160]

```
Lemma descent_set_compl n (s : {perm 'I_n.+1}) :
  descent_set (compl_perm s) = ~: descent_set s.
```

Decoded. Value-complementing every value flips ascents into descents and vice-versa, so the descent set goes to its complement.

Lemma 8.4 (β is complement-invariant)**.** $\beta_n(D) = \beta_n(\overline{D})$.

[Rocq, beta_swap.v:166]

```
Lemma beta_compl n (D : {set 'I_n}) : beta D = beta (~: D).
```

Decoded. The number of perms with descent set D equals the number with descent set $\overline{D}$, by the bijection $s \mapsto$ compl_perm(s).

Lemma 8.5 (Set-alt classification)**.** set_is_alt(D) *holds iff* $D = D^n_{\text{alt}}$ *or* $D = \overline{D^n_{\text{alt}}}$.

[Rocq, beta_swap.v:182]

```
Lemma set_is_alt_classify n (D : {set 'I_n}) :
  set_is_alt D = (D == alt_desc_set n) || (D == ~: alt_desc_set n).
```

Decoded. Exactly two sets in $'\mathtt{I}_n$ are set-alternating: the alternating pattern itself and its complement. This justifies "*the* alternating descent set" modulo the two-flavour ambiguity.

8.5 Stanley's Proposition 1.6.4

Proposition 8.6 (ω-strictness, Stanley Prop 1.6.4)**.** *If* $\omega(D) \subsetneq \omega(E)$ *then* $\beta_n(D) < \beta_n(E)$.

[Rocq, perm_seq_bridge.v:540]

```
Lemma omega_proper_beta_lt : ∀ m (D E : {set 'I_m.+1}),
  omega_set D \proper omega_set E →
  beta D < beta E.
```

Decoded. Strictly enlarging the ω-image strictly increases β. The proof goes through a deep "cd-index" machinery (min-max trees, ψ_i operators) developed in Stanley's chapter and formalized in this project's `psi_*.v` files; that material is not reproduced here.

8.6 The headline: Stanley's Corollary 1.6.5

Corollary 8.7 (Alternating set strictly maximizes β, Stanley Cor 1.6.5)**.** *For every set* $D \subseteq \{1, \ldots, n-1\}$ *that is* not *alternating,*

$$\beta_n(D) \;<\; E_n,$$

where $E_n = \beta_n(D^n_{\mathrm{alt}})$ *is the Euler number.*

[Rocq, beta_swap.v:273]

```
Lemma beta_alt_max n (D : {set 'I_n}) :
  ~~ set_is_alt D → beta D < beta (alt_desc_set n).
```

Decoded. If D is not set-alternating, then $\beta(D)$ is strictly smaller than $\beta(D^n_{\mathrm{alt}})$ — the alternating descent set is the *unique* (up to complement) maximizer of β.

Proof sketch. If D is not alternating then there exist adjacent positions $i, i+1$ both in D or both not. Toggling at i (or at a fitting nearby position) strictly enlarges $\omega(D)$, and Proposition 8.6 gives a chain of strict β-increases that terminates at D^n_{alt}. (The full chain construction uses `block_left`/`block_right` from `beta_omega.v`.) □

This is the headline theorem of the entire project. To verify axiom-freeness:

```
echo 'From mathcomp_eulerian Require Import beta_swap.
      Print Assumptions beta_alt_max.' \
  | rocq top -R . mathcomp_eulerian
```

prints `Closed under the global context.`

Chapter 9

André's reflection method (partial)

Status. This chapter records a *partial* formalization. The headline recurrence $2E_{n+2} = \sum_k \binom{n+1}{k} E_k E_{n+1-k}$ is proven modulo one named `Admitted` lemma, in the file `experimental/reflection.v` which is *out of the active build chain* (excluded from `_CoqProject`). The companion document `docs/internal/PHASE_C_RETROSPECTIVE.md` captures the eight sessions of work, the structural obstacle, and a concrete ∼510 LOC discharge plan for any future contributor.

9.1 Euler numbers

Definition 9.1 (Euler numbers)**.** $E_{n+1} = \beta_{n+1}(D^n_{\mathrm{alt}})$ (the count of alternating permutations of $\mathfrak{S}_{n+1}$). Set $E_0 = 1$.

(No `rocqsource` block since the file is out of the build chain.)

9.2 The headline recurrence

Theorem 9.2 (André recurrence, modulo one admit)**.** *For every $n \geq 0$,*

$$2E_{n+2} \;=\; \sum_{k=0}^{n+1} \binom{n+1}{k} E_k \, E_{n+1-k}.$$

The formal statement `euler_rec` in `experimental/reflection.v` (line 1705) is derived in one line from `two_eulerA_split` (already proven) and an admitted lemma:

$$\texttt{sum\_set\_is\_alt\_eq\_andre\_sum} \;:\; \sum_t \sum_p \mathrm{set\_is\_alt}(\mathrm{D}(\mathrm{insert\_max}(t,p))) \;=\; \sum_k \binom{n+1}{k} E_k E_{n+1-k}.$$

The base case $n = 0$ ($2E_2 = 2$) is proven unconditionally in the same file (`euler_rec_n0_direct`); cases $n = 1, 2$ inherit the admit transitively.

Remark 9.3 (The structural obstacle)**.** The combinatorial heart of the admit is the boundary descent at the position of the inserted maximum. After partitioning $\sigma \in \mathfrak{S}_{n+2}$ by the position p of the max and the value set L left of it, the descent at slot $p-1$ depends on (L, s_L, s_R) *jointly*: specifically on whether the largest left value (selected by s_L) exceeds the smallest right value (selected by s_R). This couples the three components, so $[\mathrm{D}(\sigma) = E]$ does not factor into independent s_L- and s_R-conditions. The classical proof absorbs the coupling via the factor of 2 on the LHS (sum over `alt` and `~:alt`), but the formal counterpart turns out to be a 4-to-1 reduction, not the 2-to-1 fibration suggested by the English proof. The retrospective explains in detail why estimates for this discharge consistently underran by a factor of ∼2–3.

Appendix A

Comprehensive lemma catalog

This appendix lists *every* named result in the active build chain — 831 entries across 29 files. Each entry shows the kind (`Definition`, `Lemma`, ...), the file:line location (clickable to GitHub), and the `coqdoc` docstring extracted from the source. Entries narrated in the main chapters of this document carry a chapter/section badge.

This catalog is generated mechanically from the `.v` files; see `docs/formal/extract_catalog.py`. Adding a docstring to a `.v` file improves both the catalog and the deployed coqdoc HTML at the same time.

`altsub.v`

`is_turn` *Definition, line 42.* [Ch 7 §7.1]
`is_turn s i` is the boolean indicator that interior position `i : 'I_n` is a turning point of `s : {perm 'I_n.+2}`: the descent indicator at slot `widen i` differs (XOR) from the indicator at slot `lift ord0 i`. These two slots straddle position `i`.

`turn_count` *Definition, line 48.* [Ch 7 §7.1]
`turn_count s` is the number of turning points of `s`, i.e., the cardinality of `[set i : 'I_n | is_turn s i]`. Stanley \S{}1.6.2: the "turn count" governing the longest alternating subsequence length.

`is_turnE` *Lemma, line 53.* [aux]
Definitional unfolding of `is_turn` to its XOR-of-descents form; used as a rewrite rule.

`alt_aux` *Fixpoint, line 73.* [aux]
`alt_aux b x xs` checks alternation of the seq `x :: xs` given the expected direction `b` of the first comparison (`true` = up, `false` = down). Successive directions flip at each step.

`is_alt` *Definition, line 83.* [Ch 7 §7.2]
`is_alt xs` holds when `xs` is alternating: comparisons of consecutive elements strictly alternate between `<` and `>`. Two starts are possible (up-down or down-up); both are accepted by disjunction.

`perm_seq` *Definition, line 97.* [Ch 7 §7.2]
`perm_seq s` is the one-line word `[s 0; s 1; ...; s (n+1)]` of `s : {perm 'I_n.+2}`, as a `seq nat`.

`size_perm_seq` *Lemma, line 101.* [aux]
`perm_seq s` always has length `n.+2`.

pick_seq *Definition, line 112.* [Ch 7 §7.2]
`pick_seq s I` is the subseq of `perm_seq s` indexed by the position set `I : {set 'I_n.+2}` in ascending order: sort `I` by underlying `val`, then read off the `s`-values.

as_perm_max *Definition, line 118.* [Ch 7 §7.3]
`as_perm_max s` is the bijective definition of `as(s)` from Stanley \S{}1.6.2: the maximum cardinality of a position set `I` whose ordered image `pick_seq s I` is alternating.

as_perm *Definition, line 124.* [Ch 7 §7.3]
`as_perm s` is the DIRECT definition (Path X) of `as(s)` as `(turn_count s).+2`. The headline `as_perm_max_eq` proves these two definitions agree.

is_alt_nil *Lemma, line 158.* [aux]
The empty seq is alternating.

is_alt_singleton *Lemma, line 162.* [aux]
A singleton seq is alternating.

alt_aux_cons *Lemma, line 172.* [aux]
Defining equation of `alt_aux` on a `cons` tail. Used as a rewrite.

is_alt_cons2 *Lemma, line 178.* [aux]
Defining equation of `is_alt` on a 2-element prefix `x :: y :: xs`: either start with an ascent or a descent. Used as a rewrite.

alt_aux_size_ge1 *Lemma, line 188.* [aux]
Shape lemma: a non-empty seq tail satisfying `alt_aux` decomposes as `y :: xs'`.

is_alt_tail *Lemma, line 196.* [aux]
Tail closure of `is_alt`: dropping the first element of an alternating seq of length at least 2 leaves an alternating seq.

turn_count_le *Lemma, line 221.* [aux]
Cardinality bound: the turn-set of `s : {perm 'I_n.+2}` has at most `n` turning points.

as_permE *Lemma, line 227.* [aux]
Definitional unfolding `as_perm s = (turn_count s).+2`; rewrite rule.

as_perm_ge2 *Lemma, line 232.* [aux]
Lower bound: any `as_perm` is at least 2 (the empty/two-element base case).

as_perm_le_size *Lemma, line 236.* [aux]
Upper bound on `as_perm`: bounded by `n.+2`, the size of `perm_seq s`.

turn_count_id *Lemma, line 241.* [aux]
The identity permutation has no descents, hence no turning points: `turn_count 1 = 0`.

as_perm_id *Lemma, line 256.* [aux]
`as_perm` hits its minimum 2 on the identity permutation.

turn_count_alt_desc *Lemma, line 275.* [aux]
When `s` has the alternating descent pattern, every interior position is a turning point: `turn_count s = n`.

as_perm_alt_desc *Lemma, line 289.* [aux]
When `s` has the alternating descent pattern, `as_perm s` attains its maximum value `n.+2`.

turn_count_max_iff *Lemma, line 295.* [aux]
Characterization of the maximum `turn_count s = n`: equivalent to every position `i : 'I_n` being a turning point.

is_alt_pick_seq_le2 *Lemma, line 322.* [aux]
Trivial alternation: any `pick_seq s I` of size at most `1` is automatically alternating.

as_perm_max_ge0 *Lemma, line 333.* [aux]
Trivial nonnegativity of `as_perm_max`; nat-valued.

as_perm_full *Lemma, line 403.* [aux]
When the descent set is alternating, `as_perm s` equals the full sequence length `size (perm_seq s) = n.+2`.

sign_seq *Definition, line 426.* [aux]
`sign_seq xs` is the boolean direction sequence at each adjacent pair: `xs[i] < xs[i+1]` for `i = 0, ..., size xs - 2`. Empty for `xs = nil`.

sign_seq_cons2 *Lemma, line 431.* [aux]
Cons reduction for `sign_seq` on a 2-prefix.

size_sign_seq *Lemma, line 436.* [aux]
Size of `sign_seq xs`: one less than `size xs`.

flip_count *Definition, line 445.* [aux]
`flip_count ss` is the number of adjacent disagreements (XOR flips) in a boolean seq `ss`. Counts indices `i` with `ss[i] != ss[i+1]`.

flip_count_cons2 *Lemma, line 451.* [aux]
Cons reduction: flip count of `a :: b :: ss` equals the boundary flip `a (+) b` plus the flip count of `b :: ss`.

turn_count_seq *Definition, line 461.* [aux]
`turn_count_seq xs` is the seq-level turn count: number of direction flips in `xs`, i.e., `flip_count (sign_seq xs)`. The seq analogue of `turn_count` for permutations.

turn_count_seq_nil *Lemma, line 464.* [aux]
`turn_count_seq` of the empty seq is `0`.

turn_count_seq_singleton *Lemma, line 468.* [aux]
`turn_count_seq` of a singleton is `0`.

turn_count_seq_pair *Lemma, line 473.* [aux]
`turn_count_seq` of a 2-element seq is `0`: only one comparison, no flip.

size_sign_seq_perm *Lemma, line 528.* [aux]
The sign sequence of `perm_seq s` has length `n.+1`, one slot per pair of consecutive positions.

enum_ord_split *Lemma, line 535.* [aux]
Decomposition of `enum 'I_n.+2` as `ord0` followed by the `lift ord0`-image of `enum 'I_n.+1`. Used to align indexing of `perm_seq` with `sign_seq`.

not_is_descentE *Lemma, line 554.* [aux]
Negated descent indicator as a strict less-than: `~~ is_descent s i` iff `val (s (widen i)) < val (s (lift ord0 i))`.

sign_seq_perm_seq *Lemma, line 571.* [aux]
Bridge identifying `sign_seq (perm_seq s)` with the (negated) descent indicator over `enum 'I_n.+1`. Connects the seq-level flip-count machinery to the perm-level descent/turn structure.

as_perm_max_le_size *Lemma, line 611.* [aux]
Trivial size bound: `as_perm_max s <= n.+2`, the size of the underlying `perm_seq s`.

sorted_val_enum_ord *Lemma, line 628.* [aux]
`enum 'I_m` is sorted by `val` ascending. Used to identify `pick_seq s [set: ...]` with `perm_seq s`.

pick_seq_setT *Lemma, line 640.* [aux]
Picking the full position set yields the full one-line word: `pick_seq s [set: 'I_n.+2] = perm_seq s`.

as_perm_max_full *Lemma, line 652.* [aux]
If `perm_seq s` is itself alternating, `as_perm_max s` hits its maximum `n.+2` (witnessed by the full set).

is_alt_size_le1 *Lemma, line 671.* [aux]
Any seq of size at most `1` is alternating.

is_alt_three *Lemma, line 676.* [aux]
An alternating seq of length at least `3` cannot have its first three entries strictly monotone in either direction.

is_alt_subseq_strictmono_le2 *Lemma, line 695.* [aux]
Strict-monotone base case for the upper bound: any alternating subseq of a strictly ascending seq has length at most 2.

slot_iv *Definition, line 730.* [aux]
`slot_iv i j` is the set of descent slots `k : 'I_n.+1` whose underlying `nat` satisfies `val i <= val k < val j`. Indexes the "slot interval" between positions `i` and `j`.

mem_slot_iv *Lemma, line 734.* [aux]
Membership in `slot_iv i j` unfolded to the `val` inequalities.

card_slot_iv *Lemma, line 740.* [aux]
Cardinality of the slot interval: `##|slot_iv i j| = val j - val i`, when `val j <= n.+1`.

slot_descent_const *Lemma, line 795.* [aux]
Constancy of the descent indicator on a turn-free slot interval: if no turning point of `s` lies in slot range `[val i, val j)`, then `is_descent s k1 = is_descent s k2` for all slots `k1, k2` in the range. Used to derive monotone behavior of `s` on turn-free zones.

constant_descent_monotone *Lemma, line 831.* [aux]
Constant-descent implies monotonicity: if `is_descent s k = b` for all slots `k` in `[val p, val q)`, then `s` is monotone on positions `(p, q)`: ascending if `b = false`, descending if `b = true`.

inter_turn_monotone *Lemma, line 903.* [aux]
Direction transport across a turn-free slot range: if no turning point lies in `[val i, val j)`, then the `s`-comparison direction on any sub-pair `(p, q)` inside `[i, j]` matches the boundary direction at `(i, j)`.

pick_flip_has_turn *Lemma, line 940.* [aux]
Existence of a witness turn for any sign flip: if three positions `i < j < k` have an `s`-comparison flip across the middle, then some turning point of `s` lies in the slot interval `[val i, val k)`.

turn_iv *Definition, line 1037.* [aux]
`turn_iv s a b` is the set of turning points `t : 'I_n` of `s` whose witness position `(val t).+1` lies in the half-open interval `[a, b)`.

mem_turn_iv *Lemma, line 1041.* [aux]
Membership in `turn_iv s a b` unfolded to its conjunction.

turn_iv_subset *Lemma, line 1046.* [aux]
`turn_iv s a b` is a subset of the global turn set of `s`.

card_turn_iv_le *Lemma, line 1051.* [aux]
Cardinality bound: `##|turn_iv s a b| <= turn_count s`.

turn_iv_split *Lemma, line 1058.* [aux]
Additivity of `turn_iv` cardinality on a split point `b`: `##|turn_iv s a c| = ##|turn_iv s a b| + ##|turn_iv s b c|` when `a <= b <= c`.

bool_triangle *Lemma, line 1156.* [aux]
Hamming-style triangle inequality on three booleans: `a XOR c <= (a XOR b) + (b XOR c)`. Building block for the flip-count monotonicity arguments.

triangle_xor_nat_g *Lemma, line 1162.* [aux]
Triangle inequality for XOR of strict nat comparisons, used to bound the flip count when an element is inserted between two others. Exploits the transitivity / total-order structure of `<` on nats.

triangle_xor_nat *Lemma, line 1180.* [aux]
Mirror variant of `triangle_xor_nat_g` with the middle position on the left. Used for "insert at front" reductions in flip count.

flip_count_pairmap_insert *Lemma, line 1199.* [aux]
Front-insertion monotonicity for `flip_count` on a `pairmap`: inserting one element at the front never decreases the flip count. Step 1 building block for `flip_count_pairmap_insert_anywhere`.

flip_step_helper *Lemma, line 1216.* [aux]
Inductive step helper for `flip_count_pairmap_insert_anywhere`: combines the IH bound `s1 <= ... + s2` with a structural matching condition to handle the four `(x < z) = (y < z)` cases uniformly.

flip_count_pairmap_insert_anywhere *Lemma, line 1251.* [aux]
Insertion-anywhere monotonicity for `flip_count` on a `pairmap`: inserting one element at any position never decreases the flip count. Generalizes `flip_count_pairmap_insert` to interior positions.

flip_count_sign_seq_insert_front *Lemma, line 1276.* [aux]
Sign-seq front-insertion monotonicity: inserting `y` at the front of `xs` never decreases `flip_count (sign_seq _)`.

subseq_drop_extra *Lemma, line 1328.* [aux]
Strict-subseq dropping: if `xs` is a proper subseq of `ys`, some index `i` of `ys` can be removed while still containing `xs`.

size_take_drop_skip *Lemma, line 1348.* [aux]
Size of ys with element at position i removed: (size ys).-1.

flip_count_pairmap_le_subseq *Lemma, line 1362.* [aux]
Subseq monotonicity for flip_count on a pairmap: if xs is a subseq of ys, its flip count is at most that of ys. Iterates flip_count_pairmap_insert_anywhere across the size difference.

flip_count_sign_seq_le_subseq *Lemma, line 1399.* [aux]
Subseq monotonicity specialized to sign_seq: if xs is a subseq of ys, then flip_count (sign_seq xs) <= flip_count (sign_seq ys). Used in as_perm_max_upper to bound subseq alternation by the global flip count.

is_alt_bool_aux *Fixpoint, line 1430.* [aux]
is_alt_bool_aux a xs checks alternation of a boolean seq with a leading "previous" element a: every adjacent pair must XOR to true.

is_alt_bool *Definition, line 1438.* [aux]
is_alt_bool xs holds when the boolean seq xs is fully alternating: every adjacent pair differs.

flip_count_is_alt_bool *Lemma, line 1446.* [aux]
Fully alternating boolean seqs maximize flip count: flip_count xs = (size xs).-1 when is_alt_bool xs holds.

sign_seq_is_alt *Lemma, line 1461.* [aux]
The sign seq of an is_alt nat seq is itself fully alternating as a boolean seq: every adjacent direction pair differs.

is_alt_flip_count *Lemma, line 1488.* [aux]
Flip-count formula for alternating seqs: an is_alt seq of size at least 2 has flip_count (sign_seq xs) = (size xs).-2. Used in as_perm_max_upper to reduce alternation to flip-count bounds.

sort_enum_strict_sorted *Lemma, line 1508.* [aux]
Sorting the enumeration of a {set 'I_n.+2} by val gives a strictly sorted seq, since the underlying enumeration is duplicate-free.

turn_inj *Definition, line 1552.* [aux]
turn_inj t injects an interior position t : 'I_n into { 'I_n.+2 } at the offset (val t).+1. Used to construct the witness set turn_witness for the lower-bound construction.

val_turn_inj *Lemma, line 1556.* [aux]
Underlying value of turn_inj t: (val t).+1.

turn_inj_inj *Lemma, line 1560.* [aux]
turn_inj is injective.

turn_witness *Definition, line 1570.* [aux]
turn_witness s is the explicit alternating-subseq witness: {ord0; ord_max} $\{}cup$ {turn_inj t | t turning point of s}. Has cardinality (turn_count s).+2; its pick_seq is the alternating subsequence of length (turn_count s).+2.

card_turn_image *Lemma, line 1574.* [aux]
Image of turn_inj on the turn set has cardinality turn_count s.

turn_image_lt_max *Lemma, line 1580.* [aux]
Range bound: any element of the `turn_inj`-image of the turn set has `0 < val x <= n`; in particular, neither `ord0` nor `ord_max`.

ord0_notin_turn_image *Lemma, line 1590.* [aux]
`ord0` is not in the `turn_inj`-image of the turn set.

ord_max_notin_turn_image *Lemma, line 1598.* [aux]
`ord_max` is not in the `turn_inj`-image of the turn set.

ord0_neq_ord_max *Lemma, line 1608.* [aux]
Trivial distinction: `ord0` and `ord_max` differ in `{'I_n.+2}`.

card_turn_witness *Lemma, line 1613.* [aux]
Cardinality of the lower-bound witness set: `##|turn_witness s| = (turn_count s).+2`.

slot_descent_const_strict *Lemma, line 1626.* [aux]
Strict-left variant of `slot_descent_const`: hypothesis only excludes turns with witness STRICTLY greater than `val i`. Used in the lower-bound construction where `i` itself may be a turn position.

inter_turn_monotone_strict *Lemma, line 1661.* [aux]
Strict-left variant of `inter_turn_monotone`: monotone direction transport with a strict lower-bound hypothesis on turn witnesses.

dir_left_of_turn *Lemma, line 1699.* [aux]
Direction LEFT of a turn witness: when `b = turn_inj t` and no interior turn lies in `(val a, val b)`, the direction `s a` vs `s b` equals `~~ is_descent s (widen_ord t)` (the "left half" descent indicator at slot `t`).

dir_right_of_turn *Lemma, line 1729.* [aux]
Direction RIGHT of a turn witness: dual of `dir_left_of_turn`; the direction `s b` vs `s c` equals `~~ is_descent s (lift ord0 t)` (the "right half" descent indicator at slot `t`).

sign_flip_at_turn *Lemma, line 1762.* [aux]
Key adjacency lemma for the lower bound: at an interior witness `b = turn_inj t` flanked by neighbors `a, c` with no interior turns in the half-open intervals, the directions across `(a, b)` and `(b, c)` DIFFER (since `t` is a turning point). Used in `triple_flip_pos_seq` to verify the alternating property of `pick_seq s (turn_witness s)`.

uniq_adj *Definition, line 1788.* [aux]
`uniq_adj xs` holds when every adjacent pair in `xs` consists of distinct elements. Weaker than `uniq`, used to characterize when `is_alt` follows from sign-seq alternation.

uniq_adj_cons2 *Lemma, line 1795.* [aux]
Cons reduction for `uniq_adj` on a 2-prefix.

uniq_adj_tail *Lemma, line 1800.* [aux]
Tail closure of `uniq_adj`.

uniq_adj_head_neq *Lemma, line 1805.* [aux]
Head distinctness extracted from `uniq_adj`.

sign_pair_neq *Lemma, line 1811.* [aux]
For distinct nats, `x < y` iff `~~ (y < x)`; trichotomy in boolean form.

alt_aux_from_sign *Lemma, line 1822*. [aux]
Key reverse direction: an alternating boolean sign seq plus adjacent distinctness implies the `alt_aux` predicate. Building block for `is_alt_from_sign`.

sign_seq_alt_of_triple_flip *Lemma, line 1850*. [aux]
Triple-flip sufficient condition for sign-seq alternation: if every adjacent triple `(js[i], js[i+1], js[i+2])` has flipping `s`-direction, then the sign seq of the `s`-image is alternating as a boolean seq.

uniq_adj_of_uniq *Lemma, line 1873*. [aux]
`uniq_adj` follows from full `uniq` for nat seqs.

is_alt_from_sign *Lemma, line 1885*. [aux]
Main reverse characterization: `is_alt xs` follows from `is_alt_bool (sign_seq xs)` together with adjacent distinctness. Used to construct alternating subseqs from sign-flip witnesses.

pos_seq *Definition, line 1908*. [aux]
`pos_seq s` is the `turn_witness s` enumerated in ascending `val` order: the sorted seq of position indices in the lower-bound witness construction.

size_pos_seq *Lemma, line 1912*. [aux]
`pos_seq s` has length `(turn_count s).+2`.

pos_seq_uniq *Lemma, line 1916*. [aux]
`pos_seq s` is duplicate-free.

pick_seq_pos_seq *Lemma, line 1921*. [aux]
Definitional unfolding: `pick_seq s (turn_witness s)` equals the map of `pos_seq s` under `val (s _)`.

pos_seq_strict_sorted *Lemma, line 1926*. [aux]
`pos_seq s` is strictly sorted by `val`.

mem_pos_seq *Lemma, line 1931*. [aux]
Membership transfer: `x \{}in pos_seq s` iff `x \{}in turn_witness s`.

pos_seq_val_lt *Lemma, line 1937*. [aux]
Strict-sorted property: index-order in `pos_seq s` matches `val`-order.

pos_seq_val_lt_inv *Lemma, line 1949*. [aux]
Converse strict-sorted property: `val`-order forces index-order in `pos_seq s`.

no_inner_in_pos_seq *Lemma, line 1962*. [aux]
Gap property: no element of `pos_seq s` has `val` strictly between consecutive entries `pos_seq s i` and `pos_seq s i.+1`.

pos_seq_nth0 *Lemma, line 1979*. [aux]
First element of `pos_seq s` is `ord0`.

pos_seq_last_val *Lemma, line 1994*. [aux]
Last element of `pos_seq s` has value `n.+1` (i.e., is `ord_max`).

interior_is_turn_inj *Lemma, line 2018*. [aux]
Interior elements of `pos_seq s` (indices `0 < i < size - 1`) are `turn_inj`-images of turning points: they correspond to actual turns of `s`.

interior_val_bounds *Lemma, line 2048.* [aux]
Strict interior value bounds: `0 < val (pos_seq s i) < n.+1` for indices `i` strictly between the endpoints.

triple_flip_pos_seq *Lemma, line 2064.* [aux]
Triple-flip property for adjacent triples in `pos_seq s`: directions across consecutive `s`-comparisons differ. The interior witness in each triple is a turning point, so `sign_flip_at_turn` applies.

is_alt_pick_turn_witness *Theorem, line 2114.* [aux]
Witness alternation: `pick_seq s (turn_witness s)` is alternating. Combines `triple_flip_pos_seq` with the sign-seq characterization of `is_alt`.

as_perm_max_lower *Theorem, line 2131.* [aux]
Headline lower bound for `as_perm_max`: `(turn_count s).+2 <= as_perm_max s`. Witnessed by `turn_witness s`, which has `(turn_count s).+2` elements and induces an alternating `pick_seq` (Stanley \S{}1.6.2).

sort_enum_subseq_enum *Lemma, line 2151.* [aux]
Sorted enumeration of a subset is a subseq of the sorted full enumeration: `sort by val (enum I)` is a subseq of `enum 'I_n.+2`.

pick_seq_subseq_perm_seq *Lemma, line 2172.* [aux]
`pick_seq s I` is a subseq of `perm_seq s` for any position set `I`.

flip_count_pick_le_perm *Lemma, line 2181.* [aux]
Flip-count comparison for `pick_seq` vs `perm_seq`: any pick has `flip_count` at most that of the full perm seq.

flip_count_as_sum *Lemma, line 2191.* [aux]
`flip_count` expressed as an indexed sum of consecutive XOR-pairs. Used to align with the `turn_count` sum over `is_turn`.

neg_add_neg *Lemma, line 2208.* [aux]
XOR is invariant under double negation: `~~ a (+) ~~ b = a (+) b`.

flip_count_perm_seq_eq_turn_count *Lemma, line 2215.* [aux]
Bridge identity: `flip_count (sign_seq (perm_seq s)) = turn_count s`. The seq-level flip count of the full word equals the perm-level turn count, via `sign_seq_perm_seq` and the descent-XOR characterization of `is_turn`.

as_perm_max_upper *Lemma, line 2252.* [aux]
Headline upper bound for `as_perm_max`: `as_perm_max s <= (turn_count s).+2`. Combines the alternating `flip_count` formula `is_alt_flip_count` with the subseq monotonicity `flip_count_pick_le_perm` and the bridge `flip_count_perm_seq_eq_turn_count`.

as_perm_max_eq *Theorem, line 2275.* [Ch 7 §7.4]
HEADLINE THEOREM (Stanley \S{}1.6.2): the longest alternating subsequence count of `s : {perm 'I_n.+2}` equals `(turn_count s).+2`. That is, `as(w) = (turn_count w).+2`, proving the two definitions `as_perm_max` (bijective max) and `as_perm` (turn count) coincide. Combines `as_perm_max_upper` and `as_perm_max_lower`.

beta.v

beta *Definition, line 22.* [Ch 5 §5.6]
`beta n D` is the set-refined descent count: the number of permutations of `'I_n.+1` whose descent set equals exactly `D`. Stanley's `beta_n(D)` (Stanley EC1, S1.4); summing `beta D` over `|D| = k` recovers `eulerian n k`.

descent_set_rev_perm *Lemma, line 31.* [aux]
Descent set of the reversal: `D(rev_perm s)` is the `rev_ord`-image of the complement of `D(s)`. Set-level refinement of `is_descent_rev`.

descent_set_rev_perm_ord *Lemma, line 44.* [aux]
The reversal permutation `n,n-1,...,0` has descent set equal to all positions.

imset_rev_ord_inv *Lemma, line 57.* [aux]
`rev_ord` is involutive on subsets: applying `imset rev_ord` twice is the identity.

imset_rev_ord_compl *Lemma, line 66.* [aux]
`rev_ord`-image commutes with set complement.

beta0 *Lemma, line 83.* [aux]
`beta_n(emptyset) = 1`: only the identity permutation has empty descent set.

descent_set_full_rev *Lemma, line 94.* [aux]
Only the strictly-decreasing permutation `rev_perm_ord` has descent set equal to all positions.

beta_full *Lemma, line 110.* [aux]
`beta_n(top) = 1`: only the reversal permutation has full descent set.

sum_beta_eq_fact *Lemma, line 124.* [aux]
`sum_D beta_n(D) = (n+1)!`: the `beta` partition of `{perm 'I_n.+1}` by descent set covers all permutations.

beta_eulerian *Lemma, line 138.* [Ch 5 §5.6]
Connection to Eulerian numbers: summing `beta D` over all `D` of cardinality `k` recovers the Eulerian number `A(n+1, k)` (Stanley EC1, S1.4).

beta_rev *Lemma, line 156.* [Ch 5 §5.6]
Reversal-complement symmetry: `beta_n(D) = beta_n(rev_ord(complement D))`. Set-level refinement of `eulerian_symm`, proved via the `rev_perm` involution.

beta_bridge.v

set_to_seq *Definition, line 29.* [aux]
`set_to_seq D` – convert the finset `D : {set 'I_n}` to the sorted `seq nat` of underlying positions; bridges the finset descent world to the seq-based `psi_cdindex` machinery.

mem_set_to_seq *Lemma, line 35.* [aux]
`mem_set_to_seq` – membership in `set_to_seq D` equals membership in the unsorted underlying value list (sorting is order-preserving on memberships).

mem_set_to_seq_iff *Lemma, line 41.* [aux]
`mem_set_to_seq_iff` – alternative form of membership in `set_to_seq D` as a `has` over `enum D`.

mem_set_to_seq_ord *Lemma, line 53.* [aux]
mem_set_to_seq_ord – ordinal-level membership bridge: val i is in set_to_seq D iff i is in D.

uniq_set_to_seq *Lemma, line 67.* [aux]
uniq_set_to_seq – the seq-of-positions set_to_seq D has no duplicates.

set_to_seq_bound *Lemma, line 76.* [aux]
set_to_seq_bound – every element of set_to_seq D is bounded above by n.

omega_seq_local *Definition, line 90.* [aux]
omega_seq_local s – the omega map at the seq level (mirrors psi_cdindex.omega_seq): the list of k in iota 0 (max s).+1 such that exactly one of k, k+1 belongs to s.

foldr_maxn_set_to_seq_lb *Lemma, line 101.* [aux]
foldr_maxn_set_to_seq_lb – any element of set_to_seq D is bounded above by foldr maxn 0 over the same list (used to ensure the iota bound in omega_seq_local covers the relevant range).

omega_set_seq_local_bridge *Lemma, line 113.* [aux]
omega_set_seq_local_bridge – key bridge: k \{}in omega_set D (finset side) iff val k \{}in omega_seq_local (set_to_seq D) (seq side). Connects the two formulations of Stanley's omega map.

beta_omega.v

sym_diff *Definition, line 27.* [Ch 8 §8.1]
sym_diff D E – symmetric difference (D \{} E) U (E \{} D) of two finsets over {set 'I_n}.

toggle_at *Definition, line 32.* [Ch 8 §8.1]
toggle_at D i – single-position toggle: flips membership of i in the descent set D by symmetric difference with [set i].

toggle_at_in *Lemma, line 37.* [aux]
toggle_at_in – membership in toggle_at D i is xor of (i == j) and (j \{}in D).

toggle_atK *Lemma, line 46.* [aux]
toggle_atK – toggling at the same position twice is the identity.

toggle_at_self *Lemma, line 55.* [aux]
toggle_at_self – at the toggled position, membership is flipped: i \{}in toggle_at D i iff i \{}notin D.

toggle_at_other *Lemma, line 61.* [aux]
toggle_at_other – away from the toggled position i, membership in D is preserved.

omega_set *Definition, line 72.* [Ch 8 §8.2]
omega_set D – Stanley's omega-map (Stanley EC1 \S{}1.6) at the finset level: k \{}in omega_set D iff exactly one of k and k+1 belongs to D (xor on consecutive positions).

mem_omega_set *Lemma, line 77.* [aux]
mem_omega_set – unfolds membership of k in omega_set D to the underlying xor of consecutive D-memberships.

toggle_at_omega_bit_i_new *Lemma, line 85.* [aux]
`toggle_at_omega_bit_i_new` – when consecutive `i,j` are both in `D`, toggling `i` turns the omega-bit at the connecting position `k` from off to on; produces the witness ordinal `k`.

toggle_at_omega_strict_superset *Lemma, line 110.* [aux]
`toggle_at_omega_strict_superset` – key omega-monotonicity step (left toggle): if `i,j` both in `D` with `j = i+1` and the predecessor of `i` also lies in `D` when defined, then toggling out `i` strictly enlarges `omega_set`. Feeds into `beta_swap` case analysis.

toggle_at_j_omega_bit_i_new *Lemma, line 154.* [aux]
`toggle_at_j_omega_bit_i_new` – right-toggle analogue of `toggle_at_omega_bit_i_new`: toggling `j` in the consecutive pair `i,j` turns the omega-bit at `k` from off to on.

toggle_at_j_omega_strict_superset *Lemma, line 180.* [aux]
`toggle_at_j_omega_strict_superset` – right-toggle omega monotonicity (case A of beta-swap): if `i,j` both in `D` with `j = i+1` and the successor of `j` is also in `D` (when defined), then toggling `j` strictly enlarges `omega_set`.

left_ok *Definition, line 224.* [aux]
`left_ok s i l` – candidate left endpoint test: `l <= i` and every position between `l` and `i` is a descent of `s`.

right_ok *Definition, line 230.* [aux]
`right_ok s i r` – candidate right endpoint test: `i <= r` and every position between `i` and `r` is a descent of `s`.

left_ok_self *Lemma, line 236.* [aux]
`left_ok_self` – a descent position `i` is its own valid left endpoint candidate.

right_ok_self *Lemma, line 245.* [aux]
`right_ok_self` – a descent position `i` is its own valid right endpoint candidate.

block_left *Definition, line 255.* [aux]
`block_left s i` – left endpoint of the maximal Foata descent block containing `i`: if `i` is a descent, the smallest `l <= i` such that all positions in `[l..i]` are descents; otherwise `i`.

block_right *Definition, line 262.* [aux]
`block_right s i` – right endpoint of the maximal Foata descent block containing `i`: if `i` is a descent, the largest `r >= i` such that all positions in `[i..r]` are descents; otherwise `i`.

block_left_left_ok *Lemma, line 268.* [aux]
`block_left_left_ok` – the chosen `block_left s i` satisfies the `left_ok` predicate (when `i` is a descent).

block_right_right_ok *Lemma, line 276.* [aux]
`block_right_right_ok` – the chosen `block_right s i` satisfies the `right_ok` predicate (when `i` is a descent).

block_left_min *Lemma, line 285.* [aux]
`block_left_min` – minimality of `block_left s i` among all valid left-endpoint candidates.

block_right_max *Lemma, line 294.* [aux]
`block_right_max` – maximality of `block_right s i` among all valid right-endpoint candidates.

block_left_le *Lemma, line 303.* [aux]
`block_left_le` – the left endpoint of the block does not exceed `i`.

block_right_ge *Lemma, line 309.* [aux]
`block_right_ge` – the right endpoint of the block is at least `i`.

block_left_descent *Lemma, line 315.* [aux]
`block_left_descent` – every position from `block_left s i` up to `i` is a descent of `s`.

block_right_descent *Lemma, line 325.* [aux]
`block_right_descent` – every position from `i` up to `block_right s i` is a descent of `s`.

block_descent_chain *Lemma, line 335.* [aux]
`block_descent_chain` – every position in the full block `[block_left s i .. block_right s i]` is a descent of `s`.

block_left_le_right *Lemma, line 350.* [aux]
`block_left_le_right` – block endpoints are ordered: `block_left s i <= block_right s i`.

block_left_minimal *Lemma, line 359.* [aux]
`block_left_minimal` – the block is left-maximal: the position immediately before `block_left s i` (when defined) is NOT a descent.

block_right_maximal *Lemma, line 384.* [aux]
`block_right_maximal` – the block is right-maximal: the position immediately after `block_right s i` is NOT a descent.

block_chain_step *Lemma, line 407.* [aux]
`block_chain_step` – single-step value comparison: for any `k` in the block, `s` strictly decreases from position `k` to `k+1`.

block_chain_values *Lemma, line 416.* [aux]
`block_chain_values` – chain-of-values: the values `s p` are strictly decreasing across the entire Foata block (from `block_left s i` through to `block_right s i + 1`).

beta_swap.v

alt_desc_set *Definition, line 34.* [Ch 8 §8.3]
`alt_desc_set n` – the canonical alternating descent set on `{set 'I_n}`: the set of even positions `{i : ~{}~{} odd i}`. The descent set that maximises `beta` (Stanley Cor 1.6.5).

mem_alt_desc_set *Lemma, line 39.* [aux]
`mem_alt_desc_set` – membership in `alt_desc_set n` reduces to parity: `i \{}in alt_desc_set n` iff `~~ odd i`.

set_is_alt *Definition, line 45.* [Ch 8 §8.3]
`set_is_alt D` – decidable predicate: `D` is "set-alternating" iff every consecutive ordinal pair has differing `D`-memberships.

alt_desc_set_is_alt *Lemma, line 51.* [aux]
`alt_desc_set_is_alt` – the canonical `alt_desc_set` satisfies `set_is_alt` (consecutive parities differ).

set_not_altP *Lemma, line 60.* [aux]
`set_not_altP` – reflection: if `D` is not set-alternating then there exist consecutive `i,j` with the same `D`-membership.

toggle_at_compl *Lemma, line 76.* [aux]
`toggle_at_compl` – the `toggle_at` operation commutes with complementation: `toggle_at (~: D) i = ~: toggle_at D i`.

alt_pair_parity *Lemma, line 94.* [aux]
`alt_pair_parity` – consecutive ordinals have opposite memberships in `alt_desc_set` (parity flips).

sym_diff_toggle_in *Lemma, line 105.* [aux]
`sym_diff_toggle_in` – toggling at any `k` in the symmetric difference `sym_diff D alt` strictly reduces the Hamming distance from `D` to `alt_desc_set n`.

sym_diff_eq0 *Lemma, line 126.* [aux]
`sym_diff_eq0` – a symmetric difference of cardinality zero forces the two sets to be equal.

compl_perm *Definition, line 145.* [Ch 8 §8.4]
`compl_perm s` – value-complement permutation: post-compose `s` with `rev_perm_ord n` so that `compl_perm s i = rev_ord (s i)`. Bijection used to prove `beta_compl` (set complement invariance).

compl_permE *Lemma, line 150.* [aux]
`compl_permE` – evaluation rule for `compl_perm s`: it sends `i` to `rev_ord (s i)`.

compl_perm_inj *Lemma, line 156.* [aux]
`compl_perm_inj` – `compl_perm` is injective on permutations (right-cancellation by `rev_perm_ord`).

compl_perm_involutive *Lemma, line 161.* [aux]
`compl_perm_involutive` – `compl_perm` is an involution: applying it twice is the identity (since `rev_perm_ord` is).

is_descent_compl *Lemma, line 172.* [aux]
`is_descent_compl` – value-complementing flips every descent bit: `is_descent (compl_perm s) i = ~~ is_descent s i`.

descent_set_compl *Lemma, line 187.* [Ch 8 §8.4]
`descent_set_compl` – value-complementing complements the descent set: `descent_set (compl_perm s) = ~: descent_set s`.

beta_compl *Lemma, line 196.* [Ch 8 §8.4]
`beta_compl` – the count `beta D` is invariant under set complementation: `beta D = beta (~: D)`. Proven via the value-complement involution `compl_perm`.

set_is_alt_classify *Lemma, line 216.* [Ch 8 §8.4]
`set_is_alt_classify` – classification of set-alternating sets (Stanley \S{}1.6): exactly two such sets exist, `alt_desc_set n` and its complement. Together with `beta_compl` this implies all set-alternating sets share the same beta value.

beta_set_is_alt_eq *Lemma, line 252.* [aux]
`beta_set_is_alt_eq` – any set-alternating `D'` has the same beta as `alt_desc_set n` (consequence of `set_is_alt_classify` and `beta_compl`).

not_set_is_alt_n_ge2 *Lemma, line 265.* [aux]
`not_set_is_alt_n_ge2` – if `D` fails `set_is_alt` then `n >= 2` (a violating consecutive pair requires at least two positions).

omega_set_alt_full *Lemma, line 277.* [aux]
`omega_set_alt_full` – the omega-set of `alt_desc_set m.+1` is the full set `[set: 'I_m]`: every consecutive position pair has differing memberships, so every omega-bit is on.

not_set_is_alt_omega_not_full *Lemma, line 295.* [aux]
`not_set_is_alt_omega_not_full` – a non-set-alternating `D` has `omega_set D` strictly smaller than the full set; the offending consecutive pair witnesses an "off" omega bit.

cycles.v

cycle_count *Definition, line 24.* [Ch 2 §2.1]
`cycle_count s` is the number of cycles in the disjoint-cycle decomposition of `s`, equivalently the cardinality of `porbits s`, the set of orbits of `s` acting on `T` by iteration. Stanley's `c(w)`.

stirling_c *Definition, line 34.* [Ch 2 §2.2]
`stirling_c n k` is the number of permutations of `'I_n` with exactly `k` cycles in their disjoint-cycle decomposition. This is Stanley's `c(n, k)`, the signless Stirling number of the first kind (Stanley EC1 \S{}1.3.2).

cycle_count_le *Lemma, line 45.* [Ch 2 §2.1]
Each cycle is non-empty, so `cycle_count s <= ##|T|`. Cycle analogue of `des_le` for descents.

porbit_id_singleton *Lemma, line 54.* [aux]
Under the identity permutation each `porbit` is the singleton `{x}`. Used in `cycle_count_id`.

cycle_count_id *Lemma, line 65.* [Ch 2 §2.1]
The identity permutation on `T` has exactly `##|T|` cycles, one singleton per element (Stanley \S{}1.3.1: each fixed point is a 1-cycle).

stirling_c_row_sum *Lemma, line 86.* [aux]
Row sum: summing `stirling_c n k` over all cycle counts `k <= n` recovers the total number of permutations of `'I_n`. Mirrors `eulerian_row_sum` in eulerian.v.

stirling_c_row_sum_fact *Lemma, line 100.* [Ch 2 §2.2]
Row sum, factorial form: `\{}sum_k c(n, k) = n!`. Stanley's identity derived from `stirling_c_row_sum` via `card_Sn`.

cycles_rec.v

lift_perm_id_iter *Lemma, line 54.* [aux]
Iterating `perm.lift_perm ord_max ord_max s` at `ord_max` stays at `ord_max`. Used in `porbit_lift_perm_id_max`.

porbit_lift_perm_id_max *Lemma, line 63.* [aux]
The `porbit` of `ord_max` under `perm.lift_perm ord_max ord_max s` is the singleton `{ord_max}`.

lift_perm_id_iter_lift *Lemma, line 75.* [aux]
Iteration on lifted points commutes with `lift ord_max`: applying `(perm.lift_perm ord_max ord_max s)^+i` at `lift ord_max k` is the lift of `(s^+i) k`.

porbit_lift_perm_id_lift *Lemma, line 85.* [aux]
The `porbit` of `lift ord_max k` under the lifted perm is the `lift ord_max`-image of the `porbit` of `k` under `s`.

porbits_lift_perm_id *Lemma, line 99.* [aux]
Decomposition of `porbits (perm.lift_perm ord_max ord_max s)` into the singleton orbit `{ord_max}` and the `lift ord_max`-images of the orbits of `s`. Used in `cycle_count_lift_perm_id`.

cycle_count_lift_perm_id *Lemma, line 123.* [aux]
H1 proper: extending `s` with `ord_max` as a fixed point adds exactly one cycle. Building block for the `j = ord_max` class of the Stirling fiber bijection (see `stirling_fiber.v`).

descent.v

is_descent *Definition, line 19.* [Ch 5 §5.1]
`is_descent s i` holds when `s` descends at position `i`, i.e. when `s i > s (i+1)`. Stanley's predicate "`i` is a descent of `w`" (Stanley EC1, S1.4).

descent_set *Definition, line 23.* [Ch 5 §5.1]
`descent_set s` is the descent set of `s`, Stanley's `D(w)` (Stanley EC1, S1.4).

des *Definition, line 26.* [Ch 5 §5.1]
`des s` is the descent number of `s`, Stanley's `d(w) = ##D(w)` (Stanley EC1, S1.4).

asc *Definition, line 29.* [Ch 5 §5.1]
`asc s = n - des s` is the ascent number of `s`.

is_descentE *Lemma, line 32.* [aux]
Equivalence with the underlying boolean for rewriting.

mem_descent_set *Lemma, line 37.* [aux]
Membership in `descent_set s` is just `is_descent s i`; rewrites `inE` form.

des_le *Lemma, line 41.* [Ch 5 §5.1]
Descent number is bounded by `n`, the number of consecutive positions.

des_add_asc *Lemma, line 45.* [aux]
Descents and ascents partition the `n` consecutive positions.

des_id *Lemma, line 49.* [Ch 5 §5.1]
The identity permutation has no descents.

is_descent_drop *Lemma, line 67.* [aux]
Descents at interior positions are preserved by `drop_perm`: for `s : {perm 'I_n.+2}` and `i : 'I_n`, the descent of `drop_perm s` at `i` corresponds to the descent of `s` at `widen_ord i`.

rev_perm_ord *Definition, line 85.* [Ch 5 §5.2]
`rev_perm_ord` is the reversal permutation `i |-> n - i` of `'I_n.+1`.

rev_perm_ordE *Lemma, line 88.* [aux]
Equivalence with the underlying `rev_ord` for rewriting.

rev_perm *Definition, line 93.* [Ch 5 §5.2]
`rev_perm s` is the reversal of `s` (compose with `rev_perm_ord` on the left). It corresponds to reading the one-line notation of `s` backwards.

rev_permE *Lemma, line 96.* [aux]
Defining equation: `rev_perm s j = s (rev_ord j)`.

is_descent_rev *Lemma, line 101.* [Ch 5 §5.2]
Reversal swaps descents and ascents pointwise: `i` is a descent of `rev_perm s` iff `rev_ord i` is not a descent of `s`.

des_rev_perm *Lemma, line 118.* [aux]
Descent count under reversal: `des (rev_perm s) = n - des s`. This is the involution underlying the symmetry `eulerian n k = eulerian n (n-k)`.

des_rev_perm_ord *Lemma, line 133.* [aux]
The reversal permutation `n,n-1,...,0` has the maximal `n` descents.

eulerian.v

eulerian *Definition, line 15.* [Ch 5 §5.3]
`eulerian n k` is the Eulerian number `A(n+1, k)`: the number of permutations of `'I_n.+1` with exactly `k` descents. Stanley's `A(n+1, k)` (Stanley EC1, S1.4).

eulerian_row_sum *Lemma, line 26.* [aux]
Row sum: summing `eulerian n k` over `k <= n` gives the cardinality of `{perm 'I_n.+1}`. Partition of permutations by descent count.

eulerian_row_sum_fact *Lemma, line 35.* [Ch 5 §5.3]
Row sum equals `(n+1)!`: combines `eulerian_row_sum` with `|S_{n+1}| = (n+1)!`.

eulerian_out_of_range *Lemma, line 40.* [Ch 5 §5.3]
Out-of-range vanishing: `eulerian n k = 0` when `k > n`, since no permutation has more than `n` descents.

des0_id *Lemma, line 48.* [Ch 5 §5.3]
Only the identity has zero descents. Used to prove `eulerian n 0 = 1`.

eulerian_n_0 *Lemma, line 78.* [aux]
Boundary value: `eulerian n 0 = 1`, counting only the identity.

rev_perm_inj *Lemma, line 91.* [aux]
Reversal is injective on permutations (left cancellation by `rev_perm_ord`).

rev_perm_involutive *Lemma, line 95.* [aux]
Reversal is an involution: `rev_perm (rev_perm s) = s`.

eulerian_symm *Lemma, line 100.* [aux]
Symmetry of Eulerian numbers: `A(n+1, k) = A(n+1, n-k)` (Stanley EC1, S1.4). Proved via the reversal involution on `{perm 'I_n.+1}`.

eulerian_n_n *Lemma, line 112.* [aux]
Boundary value: `eulerian n n = 1`, counting only the reversal permutation.

widenSn_inj *Lemma, line 132.* [aux]
`widen_ord (leqnSn n.+1) : 'I_n.+1 -> 'I_n.+2` is injective.

widenSn_neq_ord_max *Lemma, line 136.* [aux]
Widening from `'I_n.+1` to `'I_n.+2` never reaches `ord_max`.

insert_max_fun *Definition, line 151*. [Ch 5 §5.4]
Underlying function of `insert_max_perm t p`: sends `p` to `ord_max` and elsewhere lifts `t` via `widen_ord`.

insert_max_fun_p *Lemma, line 158*. [aux]
`insert_max_fun` sends position `p` to `ord_max`.

insert_max_fun_lift *Lemma, line 162*. [aux]
`insert_max_fun` on lifted positions widens `t`.

insert_max_fun_inj *Lemma, line 167*. [aux]
`insert_max_fun` is injective: combine injectivity of `widen` and `t`.

insert_max_perm *Definition, line 180*. [Ch 5 §5.4]
`insert_max_perm t p` is the permutation of `'I_n.+2` obtained from `t : {perm 'I_n.+1}` by inserting the value `ord_max` at position `p`. Underlies the bijection `{perm 'I_n.+2} ~{}= {perm 'I_n.+1} x 'I_n.+2`.

insert_max_permE *Lemma, line 183*. [aux]
Equivalence with the underlying `insert_max_fun` for rewriting.

insert_max_perm_at_p *Lemma, line 187*. [aux]
`insert_max_perm t p` sends `p` to `ord_max`, the inserted maximum value.

insert_max_perm_lift *Lemma, line 191*. [aux]
`insert_max_perm t p` on lifted positions widens `t`.

extract_max_ne *Fact, line 206*. [aux]
Off-position values of `s` avoid `ord_max` when `s p = ord_max`, by injectivity.

extract_max_fun *Definition, line 214*. [Ch 5 §5.4]
Underlying function of `extract_max_perm`: removes the value `ord_max` (located at position `p`) and reindexes the remaining values onto `'I_n.+1`.

extract_max_funE *Lemma, line 218*. [aux]
Defining equation: widening `extract_max_fun j` recovers `s (lift p j)`.

extract_max_fun_inj *Lemma, line 228*. [aux]
`extract_max_fun` is injective.

extract_max_perm *Definition, line 236*. [Ch 5 §5.4]
`extract_max_perm sp` is the permutation of `'I_n.+1` obtained from `s : {perm 'I_n.+2}` (with `s p = ord_max`) by deleting the value `ord_max`.

extract_max_permE *Lemma, line 239*. [aux]
Equivalence with the underlying `extract_max_fun` for rewriting.

extract_max_widen *Lemma, line 243*. [aux]
Defining equation: widening `extract_max_perm j` recovers `s (lift p j)`.

extract_insert_max *Lemma, line 257*. [aux]
Left inverse: extracting the max from a freshly inserted permutation recovers `t`.

insert_extract_max *Lemma, line 265*. [aux]
Right inverse: inserting the extracted max back at position `p` recovers `s`.

des_insert_max_ord0 *Lemma, line 285.* [aux]
Inserting `ord_max` at position `0` adds exactly one descent (a fresh descent is created at position `0` from the new max value).

des_insert_max_ord_max *Lemma, line 314.* [aux]
Inserting `ord_max` at position `ord_max` (the rightmost position) does not change the descent count.

des_insert_max_interior *Lemma, line 356.* [aux]
Inserting `ord_max` at an interior position above `j` adds a descent iff `j` was an ascent in `t`. Key combinatorial lemma for the Eulerian recurrence.

insert_max_perm_fiber *Lemma, line 422.* [aux]
The inverse of `insert_max_perm t p` sends `ord_max` back to `p`: identifies the fiber of position-of-max under insertion.

insert_max_perm_pair_inj *Lemma, line 430.* [aux]
The pairing map `(t, p) |-> insert_max_perm t p` is injective.

insert_max_perm_pair_surj *Lemma, line 444.* [aux]
The pairing map `(t, p) |-> insert_max_perm t p` is surjective: every permutation of `'I_n.+2` arises by inserting the max at some position.

sum_descent *Lemma, line 458.* [aux]
The descent count `des t` equals the sum of descent indicators over positions.

sum_ascent *Lemma, line 466.* [aux]
The ascent count `n - des t` equals the sum of ascent indicators over positions.

insert_max_perm_bij *Lemma, line 491.* [Ch 5 §5.4]
The pairing map `(t, p) |-> insert_max_perm t p` is a bijection `{perm 'I_n.+1} x 'I_n.+2 ~= {perm 'I_n.+2}`. Foundation of the recurrence.

eulerian_rec *Lemma, line 515.* [Ch 5 §5.5]
Eulerian recurrence (Stanley EC1, S1.4): `A(n+2, k+1) = (k+2) A(n+1, k+1) + (n+1-k) A(n+1, k)`. Proved via the insert-max bijection.

worpitzky_binom_id *Lemma, line 558.* [aux]
Key algebraic identity for Worpitzky's induction step (valid for `k <= n`): `x * C(x+k, n+1) = (k+1) C(x+k, n+2) + (n+1-k) C(x+k+1, n+2)`.

worpitzky *Lemma, line 575.* [aux]
Worpitzky's identity (Stanley EC1, S1.4): `x^(n+1) = sum_(k < n+1) A(n+1, k) * C(x+k, n+1)`. Proved by induction on `n` using `eulerian_rec` and `worpitzky_binom_id`.

binS *Lemma, line 613.* [aux]
Signed Pascal extension: a saturated-arithmetic-friendly form of Pascal's rule, valid for all `t` (including `t = 0`, where `t.-1 = 0`).

aux_id_step *Lemma, line 618.* [aux]
Key recurrence for the alternating binomial sum: `g(N+1, u) = g(N, u.-1)`. Proved via Pascal applied to both binomial factors plus a reindex.

aux_id *Lemma, line 645.* [aux]
Alternating binomial convolution identity: `sum_j (-1)^j C(n+2, j) C(t-j, n+1) = [t == n+1]`. Used in the inversion step of `eulerian_explicit`.

eulerian_explicit *Lemma, line 687.* [aux]
Eulerian explicit formula (Stanley EC1, S1.4): `A(n+1, k) = sum_(j <= k) (-1)^j C(n+2, j) (k+1-j)^(n+1)`. Proved by Worpitzky inversion against `aux_id`.

foata.v

cyc_last_to_front *Definition, line 48.* [aux]
`cyc_last_to_front s` moves the last letter of `s` to the front. Empty sequence stays empty. Building block for `foata_step`.

split_blocks_aux *Fixpoint, line 55.* [aux]
`split_blocks_aux P cur s` is the tail-recursive worker for `split_blocks`, threading the in-progress block `cur`. A boundary occurs after each `P`-letter.

split_blocks *Definition, line 67.* [aux]
`split_blocks P s` cuts `s` into maximal blocks each ending with a `P`-letter (except possibly the last block). Concatenating the blocks recovers `s`.

foata_step *Definition, line 74.* [Ch 4 §4.1]
`foata_step a u` is one step of the Foata bijection: compare the last letter of `u` to `a`, split `u` into blocks via `split_blocks` (with predicate `< a` or `a <` accordingly), cyclically rotate each block via `cyc_last_to_front`, and append `a`. See Stanley EC1 \S{}1.3.4.

foata *Definition, line 86.* [Ch 4 §4.1]
`foata w` is Foata's first fundamental bijection at the word level: iterate `foata_step` left-to-right starting from the empty word. Satisfies `inv_seq (foata w) = maj_seq w` (Theorem `foata_inv_eq_maj`).

is_desc_seq *Definition, line 95.* [Ch 4 §4.1]
`is_desc_seq w k` holds iff position `k` is a descent of `w`, i.e. `w_k > w_{k+1}`.

maj_seq *Definition, line 100.* [Ch 4 §4.1]
`maj_seq w` is the major index of `w` at the seq level: sum of 1-indexed descent positions `k+1` over `is_desc_seq w k`.

inv_seq *Definition, line 105.* [Ch 4 §4.1]
`inv_seq w` is the inversion count of `w` at the seq level: number of pairs `(i, j)` with `i < j < size w` and `w_i > w_j`.

count_gt *Definition, line 110.* [aux]
`count_gt a w` is the number of letters of `w` strictly greater than `a`.

sanity_inv_eq_maj *Lemma, line 136.* [aux]
Sanity check on Stanley's running example `w = 3,1,4,5,9,2,6`: `inv_seq (foata w) = maj_seq w = 6`.

sanity_inv_eq_maj2 *Lemma, line 146.* [aux]
Sanity check on `w = 2,3,1` (no-op fixed point of `foata`).

sanity_inv_eq_maj3 *Lemma, line 154.* [aux]
Sanity check on `w = 3,1,2`: `foata w = 1,3,2`.

cyc_last_to_front_perm_eq *Lemma, line 166.* [aux]
`cyc_last_to_front` preserves the multiset of letters.

cyc_last_to_front_size *Lemma, line 175.* [aux]
`cyc_last_to_front` preserves length.

cyc_last_to_front_uniq *Lemma, line 180.* [aux]
`cyc_last_to_front` preserves uniqueness.

split_blocks_aux_flatten *Lemma, line 186.* [aux]
Flattening the blocks of `split_blocks_aux` yields `cur ++ s`; the accumulator-aware version of `split_blocks_flatten`.

split_blocks_flatten *Lemma, line 197.* [aux]
Concatenating the blocks of `split_blocks P s` recovers `s`.

perm_eq_flatten_map_cyc *Lemma, line 203.* [aux]
Rotating each block via `cyc_last_to_front` preserves the multiset of the flattened sequence.

foata_step_perm_eq *Lemma, line 212.* [aux]
`foata_step a u` is a permutation of `rcons u a`: the step rearranges letters but preserves the multiset (with `a` inserted at the end).

foata_step_size *Lemma, line 226.* [aux]
`foata_step` grows the word length by exactly one.

foata_step_uniq *Lemma, line 234.* [aux]
`foata_step` preserves uniqueness when the new letter is fresh.

foata_perm_eq *Lemma, line 248.* [aux]
`foata w` is a permutation of `w`: the bijection preserves the multiset of letters.

foata_size *Lemma, line 263.* [aux]
`foata` preserves length.

foata_uniq *Lemma, line 267.* [aux]
`foata` preserves uniqueness.

foata_all_lt *Lemma, line 274.* [aux]
`foata` preserves any uniform upper bound on the letters.

inv_seq_rcons *Lemma, line 286.* [aux]
Classical recursion: appending letter `a` at the end of `w` increases `inv_seq` by `count_gt a w`, the number of letters of `w` above `a`.

count_gt_perm_eq *Lemma, line 316.* [aux]
`count_gt` only depends on the multiset (preserved under `perm_eq`).

maj_seq_rcons *Lemma, line 332.* [aux]
Appending letter `a` to nonempty `w` adds `size w` to `maj_seq` iff the previous last letter exceeds `a` (creating a new descent at position `size w - 1`, with 1-indexed contribution `size w`); otherwise `maj_seq` is unchanged.

foata_step_last *Lemma, line 365.* [aux]
`foata_step a u` always ends with `a`.

foata_rcons *Lemma, line 375.* [aux]
`foata` commutes with right-extension: `foata (rcons w a) = foata_step a (foata w)`. Used for induction on `w` from the right.

foata_last *Lemma, line 385.* [aux]
Last letter of `foata (a :: w)` equals last letter of `a :: w` (i.e. `last a w`).

foata_last_eq *Lemma, line 404.* [aux]
General form for the rcons induction: when `w` is non-empty, `last (foata w) = last w` for any default.

cross_inv *Definition, line 418.* [aux]
`cross_inv s1 s2` counts cross-inversions between `s1` and `s2`: pairs `(i, j)` with `s1_i > s2_j`. Equals the inversions of `s1 ++ s2` that straddle the join point.

cross_inv_nil_r *Lemma, line 422.* [aux]
`cross_inv` is zero on the right when `s2` is empty.

cross_inv_nil_l *Lemma, line 426.* [aux]
`cross_inv` is zero on the left when `s1` is empty.

cross_inv_rcons *Lemma, line 433.* [aux]
`cross_inv` recursion when an element is appended on the right.

cross_inv_cons *Lemma, line 438.* [aux]
`cross_inv` recursion when an element is prepended on the right side.

inv_seq_cat *Lemma, line 444.* [aux]
Decomposition of `inv_seq` over concatenation: left inversions, right inversions, and cross-inversions.

count_gt_cons *Lemma, line 461.* [aux]
`count_gt` recursion under cons.

count_gt_cat *Lemma, line 466.* [aux]
`count_gt` is additive under concatenation.

cross_inv_cat_l *Lemma, line 471.* [aux]
`cross_inv` distributes over concatenation in the left argument.

cross_inv_cat_r *Lemma, line 479.* [aux]
`cross_inv` distributes over concatenation in the right argument.

cross_inv_perm_eq_r *Lemma, line 484.* [aux]
`cross_inv` only depends on the multiset of its right argument.

cross_inv_perm_eq_l *Lemma, line 494.* [aux]
`cross_inv` only depends on the multiset of its left argument (since `count_gt` does).

count_lt *Definition, line 503.* [aux]
`count_lt a w` is the number of letters of `w` strictly less than `a`.

count_lt_perm_eq *Lemma, line 507.* [aux]
`count_lt` is invariant under `perm_eq`.

count_lt_cat *Lemma, line 518.* [aux]
`count_lt` is additive under concatenation.

perm_eq_flatten_map_pred *Lemma, line 524.* [aux]
If each block is `perm_eq` to its image under `f`, the flattened images are `perm_eq` to the flattened originals.

inv_seq_flatten_swap_eq *Lemma, line 537.* [aux]
Block-rotation accounting: if every block is replaced by a `perm_eq` image, the change in `inv_seq` of the flatten equals the sum of per-block changes (cross-inversions are unaffected since they only see multisets).

cyc_last_to_front_rcons *Lemma, line 570.* [aux]
Computational form: `cyc_last_to_front (rcons b' l) = l :: b'`.

inv_seq_cons_eq_rcons_shift *Lemma, line 580.* [aux]
Inversions shift when `l` moves from the end to the front of `b'`: `inv_seq (l :: b') - inv_seq (rcons b' l) = count_lt l b' - count_gt l b'`.

cyc_diff_block_lt *Lemma, line 602.* [aux]
Per-block inversion change in the `last u < a` case: rotating drops `size b'` inversions when the small letter `l < a` moves to the front past the `b' > a` block. Used by `foata_step_inv_lt`.

cyc_diff_block_gt *Lemma, line 626.* [aux]
Per-block inversion change in the `a < last u` case: rotating adds `size b'` inversions when the large letter `a < l` moves to the front past the `b' < a` block. Used by `foata_step_inv_gt`.

split_blocks_aux_all_nonempty *Lemma, line 649.* [aux]
Every block produced by `split_blocks_aux` is non-empty.

split_blocks_all_nonempty *Lemma, line 660.* [aux]
Every block produced by `split_blocks` is non-empty.

wf_block *Definition, line 678.* [aux]
`wf_block P b` holds when `b` is non-empty, its last letter satisfies `P`, and all non-last letters do not. Captures the well-formedness of blocks output by `split_blocks` (except possibly the last).

wf_block_rcons *Lemma, line 686.* [aux]
`wf_block P (rcons b' l)` is equivalent to `P l && all (~~ P) b'`.

split_blocks_aux_wf *Lemma, line 697.* [aux]
Structural invariant of `split_blocks_aux`: when `cur` is all not-`P` and the last letter of `s` satisfies `P`, every produced block is a `wf_block`.

split_blocks_wf *Lemma, line 726.* [aux]
Every block of `split_blocks P s` is a `wf_block` when `s` is non-empty and its last letter satisfies `P`.

split_blocks_aux_size_when_last_P *Lemma, line 733.* [aux]
Number of blocks of `split_blocks_aux P cur s` equals `count P s`, when `s` is non-empty and ends with a `P`-letter.

split_blocks_size_eq *Lemma, line 761.* [aux]
Number of blocks of `split_blocks P s` equals `count P s` (assuming last letter of `s` satisfies `P`).

wf_block_decomp *Lemma, line 768.* [aux]
Decomposition: any `wf_block P b` is of the form `rcons b' l` with `P l` and `b'` all not-`P`.

size_count_lt_gt *Lemma, line 783.* [aux]
For `uniq u` with `a \{}notin u`, size of `u` splits as `count_lt a u + count_gt a u` (no letter equals `a`).

sum_size_belast_wf *Lemma, line 800.* [aux]
Sum of `(size b).-1` over `wf_blocks` equals the total length minus the number of blocks.

sum_inv_cyc_lt_blocks *Lemma, line 829.* [aux]
Aggregated per-block inversion change in the `last u < a` case (`P = (y < a)`): summed cyc-rotation drop equals total internal sizes. Used by `foata_step_inv_lt`.

sum_inv_cyc_gt_blocks *Lemma, line 866.* [aux]
Aggregated per-block inversion change in the `a < last u` case (`P = (a < y)`): summed cyc-rotation gain equals total internal sizes. Used by `foata_step_inv_gt`.

split_blocks_lt_strict *Lemma, line 902.* [aux]
Strong form of `split_blocks_wf` in the `< a` case: under `uniq u` and `a \{}notin u`, every non-last letter in every block is strictly `> a` (not just not `< a`).

split_blocks_gt_strict *Lemma, line 938.* [aux]
Strong form of `split_blocks_wf` in the `a <` case: every non-last letter is strictly `< a`.

foata_step_inv_lt *Lemma, line 974.* [aux]
Case `last u < a` of `foata_step_inv`: appending `a` when last letter is smaller leaves `inv_seq` unchanged.

foata_step_inv_gt *Lemma, line 1037.* [aux]
Case `a < last u` of `foata_step_inv`: appending `a` when last letter is larger increases `inv_seq` by `size u`.

foata_step_inv *Lemma, line 1098.* [aux]
Combined per-step invariant: `foata_step` adds `size u` to `inv_seq` iff a new descent appears (i.e. `a < last u`). Mirrors `maj_seq_rcons`.

foata_inv_eq_maj *Theorem, line 1122.* [aux]
Headline equidistribution at the seq level: `inv_seq (foata w) = maj_seq w` for `uniq w`. Proven by induction on `w` from the right using `foata_step_inv` and `maj_seq_rcons`.

cyc_first_to_back *Definition, line 1154.* [aux]
`cyc_first_to_back s` moves the first letter of `s` to the back. Inverse of `cyc_last_to_front`.

cyc_first_to_back_size *Lemma, line 1161.* [aux]
`cyc_first_to_back` preserves length.

cyc_first_to_back_perm_eq *Lemma, line 1166.* [aux]
`cyc_first_to_back` preserves the multiset.

cyc_first_to_back_uniq *Lemma, line 1174.* [aux]
`cyc_first_to_back` preserves uniqueness.

cyc_first_to_backK *Lemma, line 1179.* [aux]
`cyc_first_to_back` is the left inverse of `cyc_last_to_front`.

cyc_last_to_frontK *Lemma, line 1187.* [aux]
`cyc_last_to_front` is the left inverse of `cyc_first_to_back`.

split_blocks_inv_aux *Fixpoint, line 1200.* [aux]
`split_blocks_inv_aux P cur s` is the tail-recursive worker for `split_blocks_inv`: cuts before each `P`-letter rather than after.

split_blocks_inv *Definition, line 1215*. [aux]
split_blocks_inv P s cuts s into blocks each STARTING with a P-letter (cutting right before each P-letter after the first). Inverse-direction analogue of split_blocks.

split_blocks_inv_aux_flatten *Lemma, line 1219*. [aux]
Accumulator-aware flatten property for split_blocks_inv_aux.

split_blocks_inv_flatten *Lemma, line 1232*. [aux]
Concatenating the blocks of split_blocks_inv P s recovers s.

cyc_last_to_front_wf *Lemma, line 1241*. [aux]
The rotation of a wf_block starts with a P-element and has all other elements not-P. Used by split_blocks_inv_cyc_wf.

split_blocks_inv_aux_cons_P *Lemma, line 1262*. [aux]
Cons-step for split_blocks_inv_aux when the head satisfies P: flush cur and start a new block seeded with x.

split_blocks_inv_aux_cons_notP *Lemma, line 1271*. [aux]
Cons-step for split_blocks_inv_aux when the head does not satisfy P: extend the current block by x.

split_blocks_inv_aux_app_notP *Lemma, line 1293*. [aux]
When a prefix nots is all not-P, it gets absorbed into the current block accumulator.

split_blocks_inv_aux_one_block *Lemma, line 1310*. [aux]
Feeding split_blocks_inv_aux with a wf-block-rotated l :: nots ++ tail reduces to a recursive call seeded with [:: l].

split_blocks_inv_aux_block_then_P *Lemma, line 1323*. [aux]
Single rotated block l :: nots followed by rest starting with a P-letter (or empty) produces [:: l :: nots] prepended to the recursive split of rest.

split_blocks_inv_cyc_wf *Lemma, line 1348*. [aux]
Big cancellation: applying split_blocks_inv P to the flatten of rotated wf_blocks recovers the rotated blocks themselves. Key step in proving foata_step_undoK.

foata_step_undo *Definition, line 1383*. [aux]
foata_step_undo s inverts a single foata_step: peels the trailing letter a and rotates each split_blocks_inv block back via cyc_first_to_back.

split_blocks_inv_aux_eq *Lemma, line 1394*. [aux]
split_blocks_inv_aux only depends on the predicate extensionally.

split_blocks_inv_eq *Lemma, line 1404*. [aux]
split_blocks_inv only depends on the predicate extensionally.

foata_step_undoK *Lemma, line 1413*. [aux]
Cancellation: foata_step_undo is the left inverse of foata_step on uniq inputs with a fresh letter.

foata_inv_aux *Fixpoint, line 1519*. [aux]
foata_inv_aux n s inverts n iterations of foata_step by repeatedly applying foata_step_undo.

foata_inv *Definition, line 1529*. [aux]
foata_inv s is the seq-level inverse of foata: repeatedly peels foata_steps for size s iterations.

foata_invK_aux *Lemma, line 1534.* [aux]
Length-indexed cancellation: `foata_inv_aux n (foata w) = w` for `size w = n` and `uniq w`.

foata_invK *Lemma, line 1563.* [aux]
`foata_inv` is the left inverse of `foata` on uniq sequences.

foata_inj_uniq *Lemma, line 1572.* [aux]
`foata` is injective on uniq sequences (immediate from `foata_invK`).

maj_eq_maj_seq *Lemma, line 1588.* [aux]
Bridge: perm-level `maj` equals seq-level `maj_seq` of `perm_to_seq`.

foata_perm_to_seq_size *Lemma, line 1608.* [aux]
`foata (perm_to_seq s)` has size `n.+1`; needed to build `foata_perm`.

foata_perm_to_seq_uniq *Lemma, line 1613.* [aux]
`foata (perm_to_seq s)` is uniq; needed to build `foata_perm`.

foata_perm_to_seq_bnd *Lemma, line 1618.* [aux]
`foata (perm_to_seq s)` is bounded by `n.+1`; needed to build `foata_perm`.

foata_perm *Definition, line 1628.* [Ch 4 §4.2]
`foata_perm s` is the Foata bijection lifted to `{perm 'I_n.+1}`: apply `foata` at the seq level, then re-package as a permutation.

perm_to_seq_foata_perm *Lemma, line 1633.* [aux]
Commutation: `perm_to_seq` of `foata_perm s` is `foata` of `perm_to_seq s`.

inv_eq_inv_seq *Lemma, line 1640.* [aux]
Bridge: perm-level `inv` equals seq-level `inv_seq` of `perm_to_seq`.

foata_perm_inv_maj *Lemma, line 1672.* [Ch 4 §4.3]
Headline perm-level identity: `inv (foata_perm s) = maj s`, i.e. `foata_perm` sends maj-classes to inv-classes. Direct lift of `foata_inv_eq_maj` via `maj_eq_maj_seq` and `inv_eq_inv_seq`.

foata_perm_inj *Lemma, line 1683.* [Ch 4 §4.3]
`foata_perm` is injective; follows from `foata_inj_uniq` (cancellation via `foata_invK`). On the finite set `{perm 'I_n.+1}` this implies surjective, used to prove `inv_maj_equidistr`.

inv_maj_equidistr *Theorem, line 1698.* [Ch 4 §4.3]
MacMahon's equidistribution theorem (Stanley EC1 \S{}1.3.4): the statistics `inv` and `maj` are equidistributed on `{perm 'I_n.+1}`. Proved via the bijection `foata_perm`, which satisfies `inv (foata_perm s) = maj s` and is injective hence bijective.

inversions.v

is_inv *Definition, line 27.* [Ch 3 §3.1]
`is_inv s i j` holds when the pair `(i, j)` is an inversion of `s`, i.e. `i < j` but `s i > s j`. Stanley EC1 \S{}1.3.3.

inv_set *Definition, line 32.* [Ch 3 §3.1]
`inv_set s` is the set of inversions of `s`: pairs `(i, j)` with `i < j` but `s i > s j`.

inv *Definition, line 37.* [Ch 3 §3.1]
`inv s` is the inversion count of `s`, i.e. `##|inv_set s|`. Stanley's `inv(w)`.

mem_inv_set *Lemma, line 40.* [aux]
Membership in `inv_set s` reduces to `is_inv s i j`.

inv_id *Lemma, line 45.* [Ch 3 §3.1]
The identity permutation has no inversions: `inv 1 = 0`.

inv_double_sum *Lemma, line 55.* [aux]
Bridge lemma: `inv s` expressed as the nested double-sum `\{}sum_j \{}sum_(i < j) (s j < s i)`. Bridges the `inv_set` cardinality form to the seq-level double-sum form of `inv_seq`, used in `foata.v` to prove `inv_eq_inv_seq`.

maj *Definition, line 86.* [Ch 3 §3.2]
`maj s` is the major index: the sum of the (1-indexed) descent positions of `s`. Stanley EC1 \S{}1.3.3. Our `descent_set s` is 0-indexed in `'I_n`; we shift by 1 to recover Stanley's 1-indexed positions in `{1, ..., n}`.

maj_id *Lemma, line 89.* [aux]
The identity permutation has zero major index: `maj 1 = 0`.

card_ord_pair *Lemma, line 105.* [aux]
The number of strictly-ordered pairs in `'I_m * 'I_m` is the binomial coefficient `'C(m, 2)`. Used to bound `inv` and `maj`.

inv_le *Lemma, line 146.* [Ch 3 §3.1]
Bound: `inv s <= 'C(n+1, 2)` for `s : {perm 'I_n.+1}`, since inversions are a subset of all strictly-ordered pairs.

sum_succ_ord_eq_bin2 *Lemma, line 158.* [aux]
Helper: `\{}sum_(i : 'I_n) (val i).+1 = 'C(n+1, 2)`. Used to bound `maj`.

maj_le *Lemma, line 172.* [Ch 3 §3.2]
Bound: `maj s <= 'C(n+1, 2)` for `s : {perm 'I_n.+1}`, by comparing descent-position sum to the full sum of `(val i).+1` over `'I_n`.

coinv_set *Definition, line 184.* [Ch 3 §3.1]
`coinv_set s` is the co-inversion set of `s`: pairs `(i, j)` with `i < j` and `s i < s j`. Since `s` is a permutation, every strictly-ordered pair is either an inversion or a co-inversion.

rev_ord_lt *Lemma, line 189.* [aux]
Order-reversal property of `rev_ord`: `rev_ord a < rev_ord b` iff `b < a`. Used in the `inv` reversal identity.

card_split_ord_pair *Lemma, line 198.* [aux]
Split: every strictly-ordered pair is either an inversion or a co-inversion, so `inv s + ##|coinv_set s| = 'C(n+1, 2)`.

inv_rev_perm_eq_coinv *Lemma, line 218.* [aux]
The inversions of `rev_perm s` are in bijection with the co-inversions of `s` via `(i, j) |-> (rev_ord j, rev_ord i)`.

inv_rev_perm *Lemma, line 242.* [Ch 3 §3.1]
Reversal identity for `inv`: `inv (rev_perm s) = 'C(n+1, 2) - inv s`. Stanley EC1 \S{}1.3.3.

maj_rev_perm *Lemma, line 264.* [aux]
Reversal identity for `maj`: `maj (rev_perm s) + (n+1) * des s = 'C(n+1, 2) + maj s`. The naive `maj (rev_perm s) = 'C(n+1, 2) - maj s` is FALSE; the correct identity carries a `(n+1) * des s` offset (Stanley EC1).

mmtree.v

mmtree *Inductive, line 37.* [aux]
`mmtree T` is the labelled binary tree datatype underlying Stanley's min-max tree construction `M(w)` for sequences with labels in `T`.

mmtree_to_seq *Fixpoint, line 46.* [aux]
`mmtree_to_seq t` is the in-order traversal of `t`: left subtree, then root, then right subtree.

min_pos *Definition, line 54.* [aux]
`min_pos s` is the least index `j` in `s` at which the minimum of `s` occurs; returns 0 on the empty sequence (vacuous case).

min_in *Lemma, line 59.* [aux]
`min_in` states that `foldr minn a s` always lies in the sequence `a :: s`; used to bound `min_pos` within range.

min_pos_lt *Lemma, line 72.* [aux]
`min_pos_lt` : on a nonempty sequence the split index `min_pos s` is in range, ensuring the recursion in `mmtree_of_seq_fuel` is well-defined.

mmtree_of_seq_fuel *Fixpoint, line 83.* [aux]
`mmtree_of_seq_fuel fuel s` is the fuel-bounded M1 tree construction: splits `s` at `min_pos s`, recursing on the take/drop halves. Fuel equal to `size s` suffices since each recursion strictly shrinks.

mmtree_of_seq *Definition, line 99.* [aux]
`mmtree_of_seq s` runs `mmtree_of_seq_fuel` with fuel `size s`, the M1 (minimum-only) variant of Stanley's min-max tree construction.

mmtree_of_seq_fuel_correct *Lemma, line 105.* [aux]
`mmtree_of_seq_fuel_correct` : the fuel-bounded construction round-trips in-order, i.e. `mmtree_to_seq` inverts `mmtree_of_seq_fuel` when fuel bounds the sequence size.

mmtree_of_seqK *Theorem, line 134.* [aux]
`mmtree_of_seqK` is the M1 round-trip theorem: in-order traversal inverts `mmtree_of_seq` for every input sequence.

ex_seq *Definition, line 145.* [aux]
`ex_seq` is the concrete test sequence used to demonstrate that the construction yields a genuinely branching tree.

ordinal_reindex.v

leq_lift *Lemma, line 18.* [aux]
Weak monotonicity of `lift k : 'I_n -> 'I_n.+1`: lifting preserves `<=`.

ltn_lift *Lemma, line 22.* [aux]
Strict monotonicity of `lift k : 'I_n -> 'I_n.+1`: lifting preserves `<`.

lift_image *Lemma, line 27.* [aux]
The image of `lift k : 'I_n -> 'I_n.+1` is the complement of `{k}`: `lift k` surjects onto every ordinal of `'I_n.+1` except `k` itself.

ltn_unlift_some *Lemma, line 43.* [aux]
Strict monotonicity of `unlift` where defined: if `j1, j2 != k` then the underlying ordinals returned by `unlift_some` respect the order of `j1, j2`.

perm_compress.v

drop_fun_ne *Fact, line 22.* [aux]
`s` sends `ord_max` and `lift ord_max i` to distinct values, by injectivity.

drop_fun *Definition, line 27.* [aux]
`drop_fun i` is the unique `j : 'I_n` such that `s (lift ord_max i) = lift (s ord_max) j`. This is `s` restricted to `'I_n.+1 \{} {ord_max}` and reindexed onto `'I_n`.

drop_funE *Lemma, line 30.* [aux]
Defining equation for `drop_fun`: `s (lift ord_max i)` equals `lift (s ord_max) (drop_fun i)`.

drop_fun_inj *Lemma, line 34.* [aux]
`drop_fun` is injective, inheriting injectivity from `s`.

drop_perm *Definition, line 39.* [aux]
`drop_perm s` is the permutation of `'I_n` obtained by deleting position `ord_max` from `s` and collapsing around the value `s ord_max`.

drop_permE *Lemma, line 42.* [aux]
Equivalence with the underlying `drop_fun` for rewriting.

lift_drop_permE *Lemma, line 47.* [aux]
Defining equation: lifting `drop_perm` back through `s ord_max` recovers `s` on lifted positions. Used to relate descents of `s` and `drop_perm s`.

lift_fun *Definition, line 62.* [aux]
Underlying function of `lift_perm k t`: sends `ord_max` to `k` and lifts `t` elsewhere via `lift k`.

lift_fun_ord_max *Lemma, line 69.* [aux]
`lift_fun` sends `ord_max` to `k`.

lift_fun_lift *Lemma, line 73.* [aux]
`lift_fun` on lifted positions equals `lift k (t .)`.

lift_fun_inj *Lemma, line 77.* [aux]
`lift_fun` is injective: combine injectivity of `lift` and `t`.

lift_perm *Definition, line 88.* [aux]
`lift_perm k t` is the permutation of `'I_n.+1` obtained from `t : {perm 'I_n}` by inserting a new maximum position `ord_max` sent to `k`. Inverse of `drop_perm`.

lift_permE *Lemma, line 91.* [aux]
Equivalence with the underlying `lift_fun` for rewriting.

lift_perm_ord_max *Lemma, line 95.* [aux]
`lift_perm k t` sends `ord_max` to `k`.

lift_perm_lift *Lemma, line 99.* [aux]
`lift_perm k t` on lifted positions equals `lift k (t .)`.

lift_perm_ne_k *Lemma, line 103.* [aux]
Only `ord_max` is sent to `k` by `lift_perm k t`; all other positions avoid `k`.

drop_perm_lift_perm *Lemma, line 119.* [aux]
Left inverse: dropping a freshly lifted permutation recovers the original.

lift_perm_drop_perm *Lemma, line 128.* [aux]
Right inverse: lifting a dropped permutation at the dropped value recovers the original. Establishes the bijection `{perm 'I_n.+1}` ~= `'I_n.+1` x `{perm 'I_n}`.

lift_perm_ord_max_eq *Lemma, line 136.* [aux]
Restatement of `lift_perm_ord_max` with explicit arguments, for use by clients.

perm_seq_bridge.v

perm_to_seq *Definition, line 35.* [aux]
`perm_to_seq s` – bridge from `{perm 'I_n}` to `seq nat`: the list `[val (s 0); val (s 1); ...; val (s (n-1))]`.

perm_to_seq_size *Lemma, line 40.* [aux]
`perm_to_seq_size` – the seq view of a permutation has length `n`.

perm_to_seq_uniq *Lemma, line 45.* [aux]
`perm_to_seq_uniq` – the seq view of a permutation has no duplicates (since `s` is injective).

nth_perm_to_seq *Lemma, line 53.* [aux]
`nth_perm_to_seq` – accessing `perm_to_seq s` at index `k` yields `val (s (Ordinal Hk))`.

perm_to_seq_inj *Lemma, line 63.* [aux]
`perm_to_seq_inj` – the `perm_to_seq` map is injective: distinct permutations yield distinct value lists.

is_descent_perm_seq *Lemma, line 81.* [aux]
`is_descent_perm_seq` – descent equivalence: `is_descent s i` (perm side) equals `is_descent_seq (perm_to_seq s) (val i)` (seq side).

descent_to_bvec *Definition, line 98.* [aux]
`descent_to_bvec D` – boolean-vector representation of a descent set, indexed by `enum 'I_n`: the `i`-th entry is `(i \{}in D)`.

size_descent_to_bvec *Lemma, line 102.* [aux]
`size_descent_to_bvec` – the boolean vector has length `n`.

nth_descent_to_bvec *Lemma, line 108.* [aux]
`nth_descent_to_bvec` – accessing `descent_to_bvec D` at `k` yields `(Ordinal Hk \{}in D)`.

char_mono_perm_to_seq *Lemma, line 120.* [aux]
`char_mono_perm_to_seq` – the seq-side `char_mono` of `perm_to_seq s` equals the perm-side boolean vector of `descent_set s`. Bridges `psi_cdindex` descent patterns to perm descent sets.

descent_to_bvec_inj *Lemma, line 138.* [aux]
descent_to_bvec_inj – two descent sets giving the same boolean vector must be equal.

psi_apply_psis_comm *Lemma, line 159.* [aux]
psi_apply_psis_comm – the single psi i commutes with the iterated apply_psis ss (on uniq seqs), via psi_comm.

apply_psis_rev *Lemma, line 172.* [aux]
apply_psis_rev – since each psi i is involutive and they commute, apply_psis is independent of the order of the operations: rev ss gives the same result as ss.

apply_psis_revK *Lemma, line 183.* [aux]
apply_psis_revK – applying ss then rev ss cancels back to w (each psi i is involutive).

apply_psis_cancel *Lemma, line 196.* [aux]
apply_psis_cancel – apply_psis ss is its own inverse: applying ss twice cancels back to w.

powerset_internal_apply_psis *Lemma, line 208.* [aux]
powerset_internal_apply_psis – the internal-vertex powerset used to enumerate the M-class is invariant under apply_psis; consequence of internal_vertices_apply_psis.

class_char_monos_uniq *Lemma, line 221.* [aux]
class_char_monos_uniq – the descent patterns of the M-class members of w are pairwise distinct (combines fact3 with uniq_expand_cde).

char_mono_class_inj *Lemma, line 235.* [aux]
char_mono_class_inj – M-class injectivity: within the M-class of w, two class members with the same descent pattern are actually the same sequence. Follows from fact3 and uniq_expand_cde.

desc_positions_bvec *Lemma, line 291.* [aux]
desc_positions_bvec – the descent positions extracted from the boolean vector descent_to_bvec D coincide with set_to_seq D (both sorted-asc lists of positions of D).

seq_nth_bound *Lemma, line 329.* [aux]
seq_nth_bound – under the section hypotheses (uniqness, bound, size), every entry of w read at an ordinal index is itself bounded by n.

seq_to_fun *Definition, line 337.* [aux]
seq_to_fun i – the underlying function i |-> nth 0 w (val i) on ordinals, used to lift a uniq bounded seq to a permutation.

seq_to_fun_inj *Lemma, line 341.* [aux]
seq_to_fun_inj – seq_to_fun is injective (uses uniqueness of w).

seq_to_perm *Definition, line 354.* [aux]
seq_to_perm – the permutation built from seq_to_fun; inverse of perm_to_seq for uniq, n-bounded seqs of length n.

seq_to_perm_nth *Lemma, line 358.* [aux]
seq_to_perm_nth – the value of seq_to_perm at i reads off the val i-th entry of w.

perm_to_seq_bnd *Lemma, line 370.* [aux]
perm_to_seq_bnd – every entry of perm_to_seq s is strictly less than n.

perm_to_seq_seq_to_perm *Lemma, line 377.* [aux]
perm_to_seq_seq_to_perm – round-trip identity: perm_to_seq (seq_to_perm w) = w for uniq, bounded seqs of size n.

seq_to_perm_perm_to_seq *Lemma, line 392.* [aux]
seq_to_perm_perm_to_seq – other-direction round-trip: seq_to_perm (perm_to_seq s) = s for any s : {perm 'I_n}.

all_bnd_apply_psis *Lemma, line 402.* [aux]
all_bnd_apply_psis – the boundedness predicate all (< n) is preserved by apply_psis (uses perm_eq_apply_psis).

apply_psis_size_eq *Lemma, line 412.* [aux]
apply_psis_size_eq – apply_psis preserves size: if size w = n then size (apply_psis ss w) = n.

uniq_expand_cde *Lemma, line 422.* [aux]
uniq_expand_cde – expand_cde produces a list of pairwise distinct boolean vectors (key for M-class injectivity).

nil_in_powerset_internal *Lemma, line 442.* [aux]
nil_in_powerset_internal – the empty list is always a member of powerset_internal w (the M-class always contains w itself).

char_mono_in_expand_cde *Lemma, line 455.* [aux]
char_mono_in_expand_cde – the descent pattern of w itself appears in expand_cde (phi_w w) (corresponding to ss = nil in the fact3 enumeration).

find_ss *Definition, line 475.* [aux]
find_ss w bv – search for an ss in powerset_internal w such that apply_psis ss w has descent pattern bv; returns the first match (or :: if none).

find_ss_spec *Lemma, line 481.* [aux]
find_ss_spec – whenever bv is a descent pattern realized in the M-class of w, find_ss w bv returns a valid witness: in powerset_internal w and yielding the requested pattern.

class_map *Definition, line 509.* [aux]
class_map bv sigma – map a permutation sigma to the M-class member with descent pattern bv, packaged back as a perm. The core injection used in the proof of omega_proper_beta_lt.

perm_to_seq_class_map *Lemma, line 518.* [aux]
perm_to_seq_class_map – perm_to_seq of class_map bv sigma is exactly apply_psis (find_ss ... bv) (perm_to_seq sigma).

omega_seq_mem_eq *Lemma, line 530.* [aux]
omega_seq_mem_eq – the psi_cdindex omega_seq and the local omega_seq_local (in beta_bridge) agree on memberships (definitionally identical).

omega_set_seq_bridge_bounded *Lemma, line 537.* [aux]
omega_set_seq_bridge_bounded – bridge in the bounded form: for k < m, k \{}in omega_seq (set_to_seq D) iff Ordinal Hkm \{}in omega_set D.

omega_seq_subset_bounded *Lemma, line 547.* [aux]
omega_seq_subset_bounded – a subset relation on omega-sets transports to membership at the seq level for any list of bounded indices.

index_lt_sorted *Lemma, line 568.* [aux]
index_lt_sorted – in a uniq sorted list, order on values coincides with order on positions: x < y iff index x s < index y s.

window_size_bound *Lemma, line 595.* [aux]
window_size_bound – the psi_cdindex window size at position i is bounded by size w - i.

S_w_seq_all_lt *Lemma, line 605.* [aux]
S_w_seq_all_lt – elements of S_w_seq w (the support indices of the omega map) are bounded by (size w).-2.

omega_proper_beta_lt *Lemma, line 647.* [Ch 8 §8.5]
omega_proper_beta_lt – Stanley EC1 (2nd ed.) Proposition 1.6.4 at the finset level: a strict inclusion omega_set D \{}proper omega_set E of omega-sets implies the strict inequality beta D < beta E of descent counts. Headline result of this file; proof injects {sigma | descent D} into {tau | descent E} via the M-class class_map and exhibits a strict witness via strict_witness_exists.

beta_swap_lt_caseA *Lemma, line 1129.* [aux]
beta_swap_lt_caseA – Case A of the beta-swap lemma: when i,j \{}in D with j = i+1 and the successor of j (if defined) is also in D, toggling j strictly increases beta. Combines toggle_at_j_omega_strict_superset with omega_proper_beta_lt.

psi_cdindex_core.v

window_size_apply_psis *Lemma, line 20.* [aux]
apply_psis preserves window_size at every index.

has_left_child_apply_psis *Lemma, line 29.* [aux]
apply_psis preserves has_left_child at every index.

is_internal_apply_psis *Lemma, line 38.* [aux]
apply_psis preserves is_internal.

internal_vertices_apply_psis *Lemma, line 47.* [aux]
apply_psis leaves the list of internal vertices invariant.

phi_w_apply_psis *Lemma, line 58.* [aux]
Key invariance: the cd-index phi_w is constant on the M-class generated by repeated psi applications.

phi_w_as_map *Lemma, line 72.* [aux]
phi_w w expressed as a map over internal_vertices w: each internal vertex contributes D_letter (if it has a left child) or C_letter.

leq_seqb_total *Lemma, line 89.* [aux]
Lex order on seq bool is total.

leq_seqb_trans *Lemma, line 97.* [aux]
Lex order on seq bool is transitive.

leq_seqb_anti *Lemma, line 105.* [aux]
Lex order on seq bool is antisymmetric.

sort_perm_eq_leq_seqb *Lemma, line 116.* [aux]
Sorting by `leq_seqb` is a `perm_eq`-invariant canonical form on multisets of bit-strings, used to compare M-class char_monos to `expand_cde` outputs.

descent_psi_effect *Lemma, line 131.* [aux]
Bit-level effect of `psi v` on the descent sequence at any position `k`: R-vertex toggles bit `v`; LR-vertex swaps bits `v.-1` and `v`. Underlies the char_mono-recovery proofs.

nth_char_mono *Lemma, line 178.* [aux]
`char_mono w` indexed by `k` returns the descent bit at position `k`.

size_char_mono *Lemma, line 188.* [aux]
`char_mono` has length `(size w).-1` (one bit per descent position).

is_internal_lt *Lemma, line 194.* [aux]
Internal vertices live strictly inside the descent index range, i.e. `v < (size w).-1`.

char_mono_psi_effect *Lemma, line 207.* [aux]
Bit-level effect of `psi v` on `char_mono`: a C-vertex (no left child) toggles bit at position `v`; a D-vertex (has left child) swaps bits at positions `v.-1` and `v`.

index_rcons_eq_size *Lemma, line 235.* [aux]
Helper: if `index x (rcons s y) = size s` then `x \{}notin s`.

mm_pos_rcons_lt *Lemma, line 248.* [aux]
Helper: `mm_pos` is never at the trailing position of a non-empty prefix, because that would force the last element to be both min and max, contradicting the presence of any other element.

has_left_child_t_last_valid *Lemma, line 344.* [aux]
Tree-level: in any valid mmtree, the last in-order position has no left child. Proved by structural induction on the tree.

has_left_child_last *Lemma, line 379.* [aux]
Seq-level corollary: the last vertex never has a left child.

has_left_child_last_fuel *Lemma, line 393.* [aux]
Fuel-bounded variant of `has_left_child_last`.

last_vertex_internal *Lemma, line 410.* [aux]
Penultimate vertex of a uniq seq of size at least 2 is internal: otherwise its successor (the last vertex) would have a left child, contradicting `has_left_child_last`.

has_left_child_is_internal *Lemma, line 443.* [aux]
`has_left_child` implies `is_internal` (window size > 1). Tree induction on the mmtree of `w`.

endpoint_succ_is_D_internal *Lemma, line 483.* [aux]
Every endpoint `k` with `k.+1 < size w` is the predecessor of a D-type internal vertex at `k+1` (for uniq `w`).

size_powerset_of *Lemma, line 497.* [aux]
Generic powerset-by-fold has size `2 ^ size ivs`.

size_powerset_internal *Lemma, line 515.* [aux]
`powerset_internal w` has cardinality `2 ^ |internal_vertices w|`.

uniq_powerset_of *Lemma, line 520.* [aux]
Generic powerset-by-fold is uniq when its input is uniq.

uniq_powerset_internal *Lemma, line 556.* [aux]
`powerset_internal w` is duplicate-free.

subset_powerset_of *Lemma, line 560.* [aux]
Each element of the generic powerset is a subset of the input.

subset_powerset_internal *Lemma, line 590.* [aux]
Each element of `powerset_internal w` is a subset of `internal_vertices w`.

subseq_powerset_of *Lemma, line 595.* [aux]
Each element of the generic powerset is a subseq of the input.

subseq_powerset_internal *Lemma, line 624.* [aux]
Each element of `powerset_internal w` is a subseq of `internal_vertices w`, hence is uniq when `internal_vertices w` is.

char_mono_apply_psis_C_bit *Lemma, line 631.* [aux]
Bit-level recovery for a C-vertex: bit `v` of `char_mono (apply_psis ss w)` equals the original descent bit XOR the indicator of `v \{}in ss`. Used to recover `ss` from char_monos.

char_mono_apply_psis_D_bit_pred *Lemma, line 677.* [aux]
Bit-level recovery for a D-vertex at the predecessor position `v.-1`: the bit equals the original descent bit at `v` (if `v \{}in ss`) or at `v.-1` (otherwise) – encoding the swap behaviour.

char_mono_apply_psis_D_bit_self *Lemma, line 749.* [aux]
Bit-level recovery for a D-vertex at its self position `v`: the bit equals the original descent bit at `v.-1` (if `v \{}in ss`) or at `v` (otherwise) – companion to `_D_bit_pred`.

size_expand_cde *Lemma, line 817.* [aux]
`expand_cde m` has `2 ^ k` elements, where `k` is the number of non-`E` letters in `m`.

size_phi_w *Lemma, line 831.* [aux]
`size (phi_w w)` equals the number of internal vertices of `w`.

size_expand_cde_phi_w *Lemma, line 837.* [aux]
`expand_cde (phi_w w)` has cardinality `2 ^ |internal_vertices w|`, matching the size of `powerset_internal w` for the M-class.

check_fact3 *Definition, line 854.* [aux]
`check_fact3 w` is the boolean Fact #3 statement: the multiset of M-class char_monos (sorted) equals the multiset `expand_cde (phi_w w)` (sorted).

check_fact3P *Lemma, line 863.* [aux]
Reflection between `check_fact3 w` and the equality of the sorted multisets it encodes.

check_fact3_size1 *Lemma, line 875.* [aux]
Base case for `check_fact3`: sequences of size at most 1 satisfy it trivially (no internal vertices, empty cd-index).

psi_cdindex_defs.v

is_internal *Definition, line 25.* [aux]
`is_internal i w` holds iff vertex `i` is non-endpoint in the min-max tree of `w` (its window has size > 1).

apply_psis *Definition, line 30.* [aux]
`apply_psis ops w` applies the sequence of `psi` operators left-to-right, threading `psi i` for each `i` in `ops` through `w`.

char_mono *Definition, line 35.* [aux]
`char_mono w` is the descent bit-string of `w` (length `(size w).-1`): bit `k` is `true` iff there is a descent at position `k`.

cde *Inductive, line 40.* [aux]
The cd-alphabet for classifying vertices: `C_letter` (right child only), `D_letter` (both children), `E_letter` (endpoint).

classify_vertex_cde *Definition, line 44.* [aux]
`classify_vertex_cde i w` returns the cd-letter for vertex `i` of `w`: `E_letter` if endpoint, `D_letter` if it has a left child, else `C_letter`.

phi'_w *Definition, line 51.* [aux]
`phi'_w w` is the full cde-classification string, one letter per vertex of `w` (length `size w`).

phi_w *Definition, line 56.* [aux]
`phi_w w` is the cd-index of `w`: `phi'_w w` with all `E_letter` (endpoints) stripped.

internal_vertices *Definition, line 61.* [aux]
`internal_vertices w` enumerates internal-vertex positions of `w` in ascending order.

expand_cde *Fixpoint, line 67.* [aux]
`expand_cde letters` expands a cd-word to the multiset of bit-strings: `c = a+b` yields one bit, `d = ab+ba` yields two bits in either order; `E_letter` contributes nothing.

powerset_internal *Definition, line 81.* [aux]
`powerset_internal w` enumerates all subsequences of `internal_vertices w`, used to index the M-equivalence class of `w`.

leq_seqb *Fixpoint, line 87.* [aux]
`leq_seqb` is the lexicographic order on `seq bool`, used to canonicalize the multiset of expanded cd-monomials by sorting.

apply_psis_nil *Lemma, line 99.* [aux]
Empty operator list acts as identity.

apply_psis_cons *Lemma, line 103.* [aux]
Cons unfolding: apply head `psi` then recurse.

size_apply_psis *Lemma, line 108.* [aux]
`apply_psis` preserves the length of `w`.

uniq_apply_psis *Lemma, line 115.* [aux]
`apply_psis` preserves uniqueness of `w`.

perm_eq_apply_psis *Lemma, line 122.* [aux]
`apply_psis ops w` is a permutation of `w` (each `psi` is a transposition).

apply_psis_cat *Lemma, line 129.* [aux]
Concatenation of operator lists corresponds to function composition.

apply_psis_rcons *Lemma, line 134.* [aux]
Snoc unfolding: trailing index is applied last.

psi_cdindex_support.v

S_w_seq_bound *Lemma, line 40.* [aux]
Every element of `S_w_seq w` is bounded: `k < (size w).-2`. Holds because the last vertex of `w` is always an endpoint, hence cannot contribute a `D` letter.

classify_vertex_cde_psi *Lemma, line 70.* [aux]
Vertex classification is invariant under `psi j`.

S_w_seq_psi *Lemma, line 81.* [aux]
`S_w_seq` is invariant under `psi j`, hence constant on each M-class.

mm_pos_lt_pred *Lemma, line 97.* [aux]
`mm_pos s < (size s).-1` for uniq `s` of size at least 2. Ensures the root of the min-max tree is an internal vertex.

classify_vertex_left *Lemma, line 164.* [aux]
Classification splits across `mm_pos`: for `i < j = mm_pos s`, use classification on the left subtree `take j s`.

classify_vertex_right *Lemma, line 186.* [aux]
Classification on the right side of `mm_pos`: for `i > j`, shift index by `j.+1` and classify in the right subtree `drop j.+1 s`.

classify_vertex_mm *Lemma, line 215.* [aux]
Classification at the root `j = mm_pos s` of a uniq seq of size at least 2: `D_letter` if the root has a left subtree (`j > 0`), else `C_letter`.

phi_w_cons_mm0 *Lemma, line 242.* [aux]
Cons decomposition of `phi_w` when `mm_pos = 0` (root has no left subtree): the root contributes a `C_letter` and recursion peels into the tail.

phi_w_decomp_mm *Lemma, line 272.* [aux]
Decomposition of `phi_w` at `mm_pos > 0`: `phi_w s` equals the cd-index of the left subtree concatenated with `D_letter` then the cd-index of the right subtree.

D_offsets_cons_C *Lemma, line 316.* [aux]
Cons decomposition of `D_offsets` over a `C` head: every offset shifts by 1 bit.

D_offsets_cat_D *Lemma, line 333.* [aux]
Concatenation decomposition of `D_offsets` across a `D` inserted between `m1` and `m2`: left offsets, then the `D` at offset `cde_total_width m1`, then right offsets shifted by `cde_total_width m1 + 2`.

S_w_seq_decomp_mm *Lemma, line 404.* [aux]
Decomposition of `S_w_seq` at `mm_pos > 0`, parallel to `phi_w_decomp_mm`: left positions, then `j.-1` (the D-position contributed by the root), then right positions shifted by `j.+1`.

cde_total_width_phi_w_all *Lemma, line 488.* [aux]
Total bit-width of `phi_w w` equals `(size w).-1` for any uniq `w` (proved by structural induction on size with `mm_pos` decomposition). Captures that `expand_cde (phi_w w)` elements are descent-bit strings.

cde_total_width_phi_w *Lemma, line 570.* [aux]
Sized variant of `cde_total_width_phi_w_all`; states the bit-width identity under the usable hypothesis `2 <= size w`.

D_offsets_phi_w_eq_S_w_seq *Lemma, line 585.* [aux]
Headline structural identity: the `D`-offsets of `phi_w w` coincide with `S_w_seq w`. Proved by structural induction on size with `mm_pos` decomposition.

uniq_expand_cde *Lemma, line 701.* [aux]
`expand_cde m` enumerates each cd-monomial expansion exactly once.

perm_eq_from_subset *Lemma, line 716.* [aux]
Two uniq sequences of the same size with one a subset of the other are permutations of each other.

check_fact3_of_perm_eq *Lemma, line 734.* [aux]
`perm_eq` of the M-class char_monos with `expand_cde (phi_w w)` implies the boolean Fact #3 statement.

D_vertex_descent_transition *Lemma, line 747.* [aux]
At a D-vertex (LR-internal, `has_left_child i w`), the descent bits at `i.-1` and `i` always differ – the LR-swap behaviour of `psi`.

char_mono_self_mem *Lemma, line 763.* [aux]
Self-support: `char_mono w` is itself a member of `expand_cde (phi_w w)`, using `phi_w_support_general` applied to `X = char_mono w`.

char_mono_apply_psis_mem *Lemma, line 805.* [aux]
Membership: every M-class element has its `char_mono` inside `expand_cde (phi_w w)`, by combining `phi_w_apply_psis` and `char_mono_self_mem`.

in_internal_vertices *Lemma, line 819.* [aux]
Membership in `internal_vertices` is exactly `is_internal`.

uniq_map_char_mono_powerset *Lemma, line 901.* [aux]
The list of M-class char_monos (indexed by `powerset_internal w`) is duplicate-free, by `char_mono_apply_psis_inj`.

check_fact3_true *Lemma, line 915.* [aux]
Boolean Fact #3 holds for every uniq `w`: combines uniq + same-size + membership to derive `perm_eq`, then the sorted equality.

fact3 *Lemma, line 936.* [aux]
Headline Fact #3 (Stanley EC1 \S{}1.6.3, Theorem 1.6.3): the multiset of char_monos over the M-class of `w`, canonicalised by `sort leq_seqb`, equals the canonicalised expansion `expand_cde (phi_w w)`. Direct consequence of `check_fact3_true` via `check_fact3P`.

psi_cdindex_support_defs.v

cde_width *Definition, line 20.* [aux]
cde_width l is the number of bits a single cd-letter contributes to expand_cde: C contributes 1, D contributes 2, E contributes 0.

cde_total_width *Definition, line 25.* [aux]
cde_total_width m is the bit-length of every element of expand_cde m; equals the sum of cde_width over m.

cde_offset *Definition, line 30.* [aux]
cde_offset m i is the cumulative bit-offset where the i-th letter of m starts in any element of expand_cde m.

D_offsets *Definition, line 36.* [aux]
D_offsets m enumerates the bit positions where each D in m starts; these are exactly the indices at which an expansion must have a transition (the two bits of d differ).

has_transition *Definition, line 41.* [aux]
has_transition X k holds iff bits k and k.+1 of X differ.

all_D_transitions *Definition, line 46.* [aux]
all_D_transitions m X holds iff X has a bit transition at every D-offset of m; characterizes membership in expand_cde m.

size_in_expand_cde *Lemma, line 50.* [aux]
Every element of expand_cde m has length cde_total_width m.

has_transition_cons *Lemma, line 67.* [aux]
Cons-shift for has_transition: bumping the index by 1 corresponds to dropping a leading bit.

has_transition_cons2 *Lemma, line 72.* [aux]
Cons-shift by 2 for has_transition: dropping two leading bits.

cde_offset_C_succ *Lemma, line 77.* [aux]
Cons-unfolding for cde_offset over a C head: shift by 1 bit.

cde_offset_D_succ *Lemma, line 82.* [aux]
Cons-unfolding for cde_offset over a D head: shift by 2 bits.

expand_cde_mem_transitions *Lemma, line 88.* [aux]
Forward direction of the transition characterisation: every X in expand_cde m has the required bit transitions at all D-offsets.

transitions_expand_cde_mem *Lemma, line 140.* [aux]
Backward direction: any bit-string of the right length with D-offset transitions is a member of expand_cde m.

expand_cde_mem_iff *Lemma, line 219.* [aux]
Combined characterisation: when X has the right length, membership in expand_cde m is equivalent to having a transition at every D-offset. Pivot lemma used by phi_w_support_general.

mem_filter_iota_nth *Lemma, line 234.* [aux]
Membership in the descent set filter (nth X) (iota 0 m) reduces to the bit at that index.

foldr_maxn_ge *Lemma, line 244.* [aux]
Every element of `s` is bounded above by `foldr maxn 0 s`.

mem_omega_seq *Lemma, line 254.* [aux]
Bool form of `omega_seq` membership: the XOR criterion plus the bound `k <= max s`.

has_transition_omega_seq *Lemma, line 262.* [aux]
Bridge: a bit transition in `X` at index `k < m.-1` is equivalent to `k \{}in omega_seq` of `X`'s descent set. Pivot used in the support proof to translate D-offset transitions into `omega_seq` membership.

cde_total_width_cat *Lemma, line 294.* [aux]
`cde_total_width` is additive over concatenation.

psi_cdindex_tree.v

window_size_last_fuel *Lemma, line 44.* [aux]
Fuel-bounded variant: `window_size` at the last index of a non-empty `w` equals 1. Used to prove the leaf-status of the last vertex.

window_size_last *Lemma, line 73.* [aux]
Last vertex has window size 1, i.e. is always an endpoint.

window_size_t *Fixpoint, line 85.* [aux]
`window_size_t i t` is the tree-level analogue of `window_size`: recursion on the `mmtree t` using the in-order index `i`.

has_left_child_t *Fixpoint, line 96.* [aux]
`has_left_child_t i t` is the tree-level analogue of `has_left_child`: direct structural recursion on the `mmtree`.

is_internal_t *Definition, line 106.* [aux]
Tree-level analogue of `is_internal`.

window_size_t_Node_lt *Lemma, line 115.* [aux]
Cons-style equation: index in left subtree forwards to `l`.

window_size_t_Node_eq *Lemma, line 121.* [aux]
At the root index, the window covers the root and the right subtree.

window_size_t_Node_gt *Lemma, line 127.* [aux]
Index above the root forwards to the right subtree with shifted index.

has_left_child_t_Node_lt *Lemma, line 141.* [aux]
Cons-style equation for `has_left_child_t`: index in left subtree.

has_left_child_t_Node_eq *Lemma, line 147.* [aux]
Root has a left child iff the left subtree is non-empty.

has_left_child_t_Node_gt *Lemma, line 153.* [aux]
Index above the root forwards `has_left_child_t` to right subtree.

size_mmtree_to_seq_node *Lemma, line 167.* [aux]
In-order seq of `Node l x r` has size `|l| + |r| + 1`.

has_left_child_t_0 *Lemma, line 173.* [aux]
Position 0 (leftmost vertex) never has a left child.

window_size_t_eq *Lemma, line 188.* [aux]
Bridge: tree-level `window_size_t` agrees with seq-level `window_size` on the in-order traversal of any valid mmtree.

has_left_child_t_eq *Lemma, line 207.* [aux]
Bridge: tree-level and seq-level `has_left_child` agree.

is_internal_t_eq *Lemma, line 224.* [aux]
Bridge: tree-level and seq-level `is_internal` agree.

endpoint_implies_next_t *Lemma, line 236.* [aux]
Tree-level: an endpoint at position `k` is immediately followed (at `k.+1`) by a vertex with a left child. Used to bound `S_w_seq`.

LR_pred_is_endpoint_t *Lemma, line 303.* [aux]
Tree-level: the predecessor of an LR-vertex (D-vertex) is an endpoint. Key structural property used in the bit-recovery lemmas.

endpoint_implies_next_has_left_child *Lemma, line 393.* [aux]
Seq-level corollary of `endpoint_implies_next_t` via the mmtree bridge.

LR_pred_is_endpoint *Lemma, line 411.* [aux]
Seq-level corollary of `LR_pred_is_endpoint_t`: D-vertex predecessor is always an endpoint. Used by the bit-recovery proofs in `psi_cdindex_core.v`.

psi_cdindex_tree_shape.v

mmtree_shape_fuel *Fixpoint, line 29.* [aux]
`mmtree_shape_fuel fuel s` is the fuel-bounded shape encoding of the min-max tree of `s`: head = `mm_pos s`, tail = shape of left subtree concatenated with shape of right subtree.

mmtree_shape *Definition, line 48.* [aux]
`mmtree_shape s` is the canonical shape encoding of the min-max tree of `s`, obtained by saturating the fuel at `size s`. Two sequences with equal shape have isomorphic min-max trees.

mmtree_shape_fuel_monotone *Lemma, line 54.* [aux]
`mmtree_shape_fuel` is independent of the fuel once it is large enough.

mmtree_shape_nil *Lemma, line 78.* [aux]
Shape of the empty sequence is empty.

mmtree_shape_cons *Lemma, line 83.* [aux]
Cons unfolding: shape of `a :: s0` = `mm_pos` then shapes of left and right subtrees, mirroring the tree-construction recursion.

size_mmtree_shape *Lemma, line 106.* [aux]
Shape encoding has the same length as the underlying sequence.

mmtree_shape_order_iso *Lemma, line 134.* [aux]
Headline structural lemma: order-isomorphic sequences (same size, both uniq, same comparison pattern) produce the same tree shape. This is the single heavy proof underlying the per-property invariance lemmas below.

seq_cat_left_eq *Lemma, line 189.* [aux]
Left-cancellation for `++` when prefixes have equal size.

seq_cat_right_eq *Lemma, line 200.* [aux]
Right-cancellation for ++ when prefixes have equal size.

mmtree_shape_decompose *Lemma, line 212.* [aux]
Inverse of mmtree_shape_cons: equal shapes split into equal roots and equal subtree shapes, providing a structural recursion principle.

has_left_child_of_shape *Lemma, line 244.* [aux]
has_left_child depends only on the tree shape: equal shapes give equal has_left_child at every index.

window_size_of_shape *Lemma, line 291.* [aux]
window_size depends only on the tree shape.

has_left_child_order_iso *Lemma, line 345.* [aux]
Compatibility wrapper used downstream: order-isomorphic uniq sequences give equal has_left_child at every index. Composes has_left_child_of_shape with mmtree_shape_order_iso.

mmtree_shape_psi *Lemma, line 367.* [aux]
Heavy lemma: applying psi j does not change the tree shape (for uniq w). Proved by case-splitting on j vs m = mm_pos w and reducing the root case to mmtree_shape_order_iso on the rotated tail.

has_left_child_psi *Lemma, line 485.* [aux]
Headline corollary: psi j preserves has_left_child. Five-line proof composing has_left_child_of_sha
with mmtree_shape_psi.

psi_cdindex_witness.v

omega_seq *Definition, line 30.* [aux]
omega_seq s is the seq-level omega map: positions k such that exactly one of k, k+1 belongs to s. Mirrors omega_set from beta_swap.v on raw seq nat (no finset).

is_D_letter *Definition, line 39.* [aux]
is_D_letter l tests whether l is the D cd-letter.

S_w_seq *Definition, line 45.* [aux]
S_w_seq w is the d-position set of w: predecessors of D-vertices in the cd-classification. This is the support set S_w from Stanley's Prop 1.6.4.

check_phi_w_support *Definition, line 60.* [aux]
Boolean form of the support claim: X expands from phi_w w iff every d-position of w is in omega of X's descent set.

omega_monotone_class_count *Lemma, line 107.* [aux]
Weak monotonicity: if omega_seq S is contained in omega_seq T, then every M-class whose d-positions are covered by omega_seq S is also covered by omega_seq T. Half of Stanley's Prop 1.6.4.

witness_perm *Definition, line 128.* [aux]
witness_perm n k is the permutation 1;..;k; k+2; k+1; k+3;..;n: a single inversion at position k embedded in a sorted backbone. Realises a cd-word with S_w = {k}, used in strict_witness_exists.

leq_maxn *Lemma, line 132.* [aux]
Local arithmetic helper: maxn a b = a when b <= a.

geq_maxn *Lemma, line 140.* [aux]
Local arithmetic helper: `maxn a b = b` when `a <= b`.

geq_minn *Lemma, line 144.* [aux]
Local arithmetic helper: `minn a b = a` when `a <= b`.

path_leq_last *Lemma, line 152.* [aux]
A leq-path from `a` over `s` gives `a <= last a s`.

foldr_maxn_path *Lemma, line 161.* [aux]
On a sorted (leq-path) sequence, `foldr maxn` reduces to its last element.

foldr_minn_ge *Lemma, line 181.* [aux]
`foldr minn a s = a` when every element of `s` is at least `a`.

all_le_iota *Lemma, line 190.* [aux]
Every element of `iota m.+1 n` is at least `m`.

min_pos_iota *Lemma, line 197.* [aux]
`min_pos` of an ascending `iota` is 0 (minimum is the head).

path_iota *Lemma, line 204.* [aux]
`iota m.+1 n` is a leq-path with starting point `m`.

last_iota *Lemma, line 208.* [aux]
Last element of `m :: iota m.+1 n` is `m + n`.

foldr_maxn_iota *Lemma, line 215.* [aux]
`foldr maxn` of an ascending `iota` equals its last element.

max_pos_iota *Lemma, line 225.* [aux]
`max_pos` of an ascending `iota` is the last index.

mm_pos_iota *Lemma, line 242.* [aux]
`mm_pos` of an ascending `iota` is 0 (root is the minimum).

size_witness_perm *Lemma, line 248.* [aux]
`witness_perm n k` has length `n` when `k + 2 < n`.

iota_mem_range *Lemma, line 258.* [aux]
Membership in `iota m l` gives `m <= i < m + l`.

witness_perm_uniq *Lemma, line 262.* [aux]
`witness_perm n k` is duplicate-free when `k + 2 < n`.

check_strict_witness *Definition, line 303.* [aux]
Boolean witness: `witness_perm n k` is uniq, of length `n`, and has `S_w_seq` equal to `[:: k]`. Used for vm_compute sanity checks.

check_strict_witness_correct *Lemma, line 329.* [aux]
`check_strict_witness` reflects `strict_witness_exists` for that `n,k`.

has_left_child_iota *Lemma, line 341.* [aux]
Helper: ascending sequences (`iota`) have no left children in their min-max tree, since `mm_pos = 0` at every recursion depth.

foldr_minn_all_gt *Lemma, line 354.* [aux]
Helper: `foldr minn a s = a` when `a` is strictly less than every element of `s`.

mm_pos_min_first *Lemma, line 365.* [aux]
`mm_pos (a :: s) = 0` when `a` is the strict minimum of `a :: s`.

min_pos_core *Lemma, line 378.* [aux]
`min_pos` of the witness core `[k+2; k+1] ++ iota k.+3 m` is 1 (the `k+1` at position 1 is the unique minimum).

max_pos_core_gt0 *Lemma, line 392.* [aux]
`max_pos` of the witness core is positive when the suffix `iota k.+3 m` is non-empty.

mm_pos_core *Lemma, line 407.* [aux]
`mm_pos` of the witness core is 1 (position of `k+1`) when the suffix is non-empty.

hlc_core_not1 *Lemma, line 424.* [aux]
In the witness core, only position 1 has a left child; all other positions are leaves of the min-max tree.

hlc_core_1 *Lemma, line 443.* [aux]
Position 1 in the witness core has a left child (the `k+2` vertex).

ws_core_1 *Lemma, line 456.* [aux]
`window_size` at the `mm_pos` position 1 in the witness core is `m.+1` (covers `k+1` and the entire ascending suffix).

S_w_seq_core *Lemma, line 472.* [aux]
`S_w_seq` of the witness core is `[:: 0]`: the only D-position is at index 1, contributing `0` to the d-position set.

S_w_seq_witness_k0 *Lemma, line 508.* [aux]
Base case `k = 0`: `witness_perm n 0` is exactly the witness core, so `S_w_seq = [:: 0]`.

classify_skip_mm0 *Lemma, line 521.* [aux]
When the head `a` is at the root (`mm_pos = 0`), classifying any positive position `i` in `a :: rest` reduces to classifying `i-1` in the right subtree `rest`.

S_w_seq_shift *Lemma, line 544.* [aux]
Index-shift property: when the head is at the root, `S_w_seq` of the cons equals the `S_w_seq` of the tail with every entry incremented.

drop1_witness_map_succ *Lemma, line 587.* [aux]
Tail of `witness_perm n k.+1` equals the `+1`-shift of `witness_perm n.-1 k`; provides the inductive step for k.

map_succ_order_iso *Lemma, line 610.* [aux]
Mapping `S` over a uniq sequence preserves the order of elements (used to lift order-iso properties through the witness shift).

classify_map_succ *Lemma, line 622.* [aux]
`classify_vertex_cde` is invariant under the `+1`-shift on a uniq sequence (since cd-classification depends only on order).

S_w_seq_map_succ *Lemma, line 650.* [aux]
`S_w_seq` is invariant under the `+1`-shift on a uniq sequence.

mm_pos_witness *Lemma, line 663.* [aux]
`mm_pos` of `witness_perm n k` is 0 for `k >= 1`: the `1` at the head is the strict minimum of the permutation.

strict_witness_exists *Lemma, line 714.* [aux]

Strict-witness existence (Stanley Prop 1.6.4, strict half): for every `k < n.-2` there is a uniq permutation `w` of length `n` with `S_w_seq w = [:: k]`. The witness is `witness_perm n k`.

psi_comm.v

sorted_uniq_nth_ltn *Lemma, line 15.* [aux]

`sorted_uniq_nth_ltn` : on a uniq sorted sequence, value order at two indices is the same as index order; ranking helper for `psi_comm`.

shift_preserves_ltn *Lemma, line 36.* [aux]

`shift_preserves_ltn` : the modular shift by `delta` preserves the `<` relation on ranks within the appropriate non-extremum sub-range. Modular-arithmetic core of interior order preservation.

rank_shift_preserves_interior_order *Lemma, line 79.* [aux]

`rank_shift_preserves_interior_order` : at non-head positions `p, q` the rank-shift on `L` is order-preserving (no wrap-around). Used to propagate descent comparisons through `psi`.

window_size_psi *Lemma, line 163.* [aux]

`window_size_psi` : applying `psi j` does not change `window_size i` at any other position `i` – generalization of `window_size_psi_self` to cross positions, foundational for `psi_comm`.

window_at_psi_disjoint *Lemma, line 324.* [aux]

`window_at_psi_disjoint` : `psi j` preserves `window_at i` verbatim when the windows at `i` and `j` are disjoint (i.e. `i + ws_i <= j`).

psi_comm_disjoint_lr *Lemma, line 354.* [aux]

`psi_comm_disjoint_lr` : disjoint commutativity in the directed form `i + ws_i <= j`; combinatorially trivial since the windows act on non-overlapping slices of `w`.

psi_comm_disjoint *Lemma, line 446.* [aux]

`psi_comm_disjoint` : symmetric form of `psi_comm_disjoint_lr` – `psi i` and `psi j` commute whenever the two M-windows are disjoint.

window_size_psi_ancestor *Lemma, line 479.* [aux]

`window_size_psi_ancestor` : nested specialization of `window_size_psi` – when `W_j` sits inside `W_i`, applying `psi i` preserves `window_size j`.

sort_map_mono *Lemma, line 513.* [aux]

`sort_map_mono` : `sort leq` commutes with a monotone injection on the elements of `L`; structural helper for `psi_map_comm`.

index_map_inj_in *Lemma, line 551.* [aux]

`index_map_inj_in` : `index` commutes with a locally injective map `f`, a structural helper for the rank-shift transfer.

mono_inj_in *Lemma, line 574.* [aux]

`mono_inj_in` : a `<`-monotone function on a finite set is injective there; trivial corollary used in `psi_map_comm`.

rank_shift_map_comm *Lemma, line 590.* [aux]

`rank_shift_map_comm` : `rank_shift_seq` commutes with any monotone injection on the elements of `L`; algebraic core of `psi_map_comm`.

psi_map_comm *Lemma, line 657.* [aux]
psi_map_comm : psi k commutes with any comparison-preserving relabelling f. Reduces nested commutativity to interior-only rank-shift commutativity.

rank_shift_psi_comm *Lemma, line 720.* [aux]
rank_shift_psi_comm : when d is rooted at the head (mm_pos d = 0) and k > 0, rank_shift_seq commutes with psi k. Algebraic core of psi_comm_nested.

psi_comm_nested *Lemma, line 832.* [aux]
psi_comm_nested : nested commutativity – when W_j sits inside W_i, psi i and psi j commute. Combines window-stability and rank_shift_psi_comm.

psi_comm *Theorem, line 1348.* [aux]
psi_comm is the M3 commutativity theorem (Stanley Fact #1, half 1): the operators psi i all pairwise commute on uniq inputs. Proved by window_trichotomy dispatching to disjoint or nested commutativity.

psi_core.v

max_pos *Definition, line 18.* [aux]
max_pos s is the least index in s at which the maximum of s occurs, the dual of min_pos.

mm_pos *Definition, line 23.* [aux]
mm_pos s is Stanley's min-or-max split index: the least index at which either the minimum or the maximum of s occurs.

max_in *Lemma, line 28.* [aux]
max_in is the dual of min_in: foldr maxn a s always lies in a :: s.

max_pos_lt *Lemma, line 39.* [aux]
max_pos_lt : on a nonempty sequence the index max_pos s is in range.

mm_pos_lt *Lemma, line 47.* [aux]
mm_pos_lt : on a nonempty sequence the min-or-max split index is in range, ensuring mmtree_of_seq_mm_fuel recursion is well-defined.

mmtree_of_seq_mm_fuel *Fixpoint, line 55.* [aux]
mmtree_of_seq_mm_fuel fuel s is the Stanley-correct min-max tree construction (M2 variant): splits s at mm_pos s rather than min_pos s, producing the alternating-extremum tree shape.

mmtree_of_seq_mm *Definition, line 71.* [aux]
mmtree_of_seq_mm s is Stanley's min-max tree M(w) for s = w: the canonical M-class representative used to define psi.

mmtree_of_seq_mm_fuel_correct *Lemma, line 76.* [aux]
mmtree_of_seq_mm_fuel_correct : the M2 fuel construction round-trips in-order, the analogue of mmtree_of_seq_fuel_correct.

mmtree_of_seq_mmK *Theorem, line 100.* [aux]
mmtree_of_seq_mmK is the M2 round-trip theorem: in-order traversal inverts the Stanley min-max construction mmtree_of_seq_mm.

window_size_fuel *Fixpoint, line 114.* [aux]
window_size_fuel fuel i s computes the size of the M-window at in-order position i (i.e. 1 + |right subtree of vertex i|) by fuel-bounded recursion mirroring mmtree_of_seq_mm_fuel.

window_size *Definition, line 131.* [aux]
`window_size i s` is `1 + |right subtree of vertex at in-order position i|` in `M(s)`, i.e. the size of the M-window on which `psi i` acts; `0` when `i >= size s`.

window_at *Definition, line 137.* [aux]
`window_at i w` is the in-order labels of the M-window at position `i`: the contiguous slice `w_i, w_{i+1}, ..., w_{i+ws-1}` where `ws = window_size i w`.

window_size_fuel_bound *Lemma, line 148.* [aux]
`window_size_fuel_bound` : combined invariant for the fuel version – the window is positive iff `i` is in range, and is bounded by the available suffix length.

window_size_gt0 *Lemma, line 208.* [aux]
`window_size_gt0` : the M-window at an in-range position is nonempty.

window_size_bound *Lemma, line 217.* [aux]
`window_size_bound` : the M-window at `i` never exceeds the suffix `size w - i`, so `window_at i w` is a slice of `w`.

window_size_oor *Lemma, line 224.* [aux]
`window_size_oor` : out-of-range positions have empty window, by which `psi i` degenerates to the identity.

rank_shift_seq *Definition, line 241.* [aux]
`rank_shift_seq L` is the rank-shift on the M-window: replace the head with the opposite extremum (max if head = min, else min) and rotate the other ranks accordingly. This is the algebraic core of `psi_i`.

size_rank_shift_seq *Lemma, line 249.* [aux]
`size_rank_shift_seq` : `rank_shift_seq` preserves length (definitional).

psi *Definition, line 261.* [aux]
`psi i w` is Stanley's `psi_i` operator on the M-class representative `w`: it fixes the prefix of length `i`, applies the rank-shift to the M-window `window_at i w`, and concatenates the unchanged suffix. Total function; identity outside the window or when `i >= size w`.

map_nth_iota_sorted *Lemma, line 268.* [aux]
`map_nth_iota_sorted` : mapping `nth 0 sorted` across `iota 0 (size sorted)` reproduces `sorted`. Helper for the rank-shift permutation analysis.

map_mod_iota_rot *Lemma, line 282.* [aux]
`map_mod_iota_rot` : the modular shift `r |-> (r + shift) %% k` on `iota 0 k` equals a rotation of `iota 0 k`; key step in proving `rank_shift_perm_eq`.

rank_shift_perm_eq *Lemma, line 316.* [aux]
`rank_shift_perm_eq` : `rank_shift_seq L` is a permutation of `L`, so `psi i` preserves the multiset of labels in `w`.

sort_rank_shift_seq *Lemma, line 357.* [aux]
`sort_rank_shift_seq` : the rank-shift commutes with `sort leq`, a consequence of `rank_shift_perm_eq`.

uniq_rank_shift_seq *Lemma, line 370.* [aux]
`uniq_rank_shift_seq` : the rank-shift preserves uniqueness, by permutation.

size_rank_shift_seq2 *Lemma, line 377.* [aux]
size_rank_shift_seq2 : size preservation derived from perm_eq, convenient form for downstream rewrites.

psi_perm_eq *Lemma, line 385.* [aux]
psi_perm_eq is the M2 multiset preservation theorem: psi i w is a permutation of w for every i and w.

rank_shift_seqE *Lemma, line 418.* [aux]
rank_shift_seqE : explicit map characterization of rank_shift_seq L in terms of the sort and the head's extremum role.

nth_rank_shift_seq *Lemma, line 434.* [aux]
nth_rank_shift_seq : positionwise formula for rank_shift_seq L, used in the involutivity and interior-order proofs.

head_rank_shift_seq *Lemma, line 447.* [aux]
head_rank_shift_seq : the head of rank_shift_seq L is the formula above instantiated at n = 0.

rank_shift_seq_involutive *Lemma, line 462.* [aux]
rank_shift_seq_involutive : applying the rank-shift twice is the identity, provided the head of L is the global min or max. Algebraic heart of the psi_involutive theorem.

window_size_fuel_monotone *Lemma, line 535.* [aux]
window_size_fuel_monotone : the fuel-bounded window_size is independent of fuel as long as fuel covers the input size.

window_size_cons *Lemma, line 563.* [aux]
window_size_cons : structural unfolding of window_size on a cons by the mm_pos split, mirroring the recursion of mmtree_of_seq_mm.

psi_id_oor *Lemma, line 587.* [aux]
psi_id_oor : psi i w = w when i is out of range, the edge convention from M2_PSI_INFORMAL.md.

psi_id_trivial *Lemma, line 598.* [aux]
psi_id_trivial : psi i w = w on a trivial window (size <= 1, i.e. a leaf or single-vertex right subtree).

window_at_cons *Lemma, line 619.* [aux]
window_at_cons : structural unfolding of window_at on a cons, by the mm_pos split. Companion of window_size_cons.

size_psi *Lemma, line 660.* [aux]
size_psi : psi i preserves length, derived from psi_perm_eq.

uniq_psi *Lemma, line 664.* [aux]
uniq_psi : psi i preserves uniqueness, derived from psi_perm_eq.

take_psi *Lemma, line 669.* [aux]
take_psi : psi j does not touch positions strictly before j, so any prefix of length k <= j is unchanged.

foldr_minn_aux *Lemma, line 693.* [aux]
foldr_minn_aux : foldr minn a s bounds a and every element of s from above; technical helper for the extremum reasoning.

foldr_minn_le *Lemma, line 706.* [aux]
`foldr_minn_le` : convenient form of `foldr_minn_aux` – the fold-min bounds every element of `a :: s` from below.

foldr_maxn_aux *Lemma, line 713.* [aux]
`foldr_maxn_aux` : dual of `foldr_minn_aux` for `maxn`.

foldr_maxn_ge *Lemma, line 725.* [aux]
`foldr_maxn_ge` : dual of `foldr_minn_le` for `maxn`.

min_val_perm_eq *Lemma, line 733.* [aux]
`min_val_perm_eq` : the global min as computed by `foldr minn` depends only on the multiset, so `psi i` preserves the minimum value.

max_val_perm_eq *Lemma, line 746.* [aux]
`max_val_perm_eq` : dual of `min_val_perm_eq` for the global max.

window_fits_left *Lemma, line 762.* [aux]
`window_fits_left` : when `i` sits in the left subtree of the root (`i < mm_pos w`), the M-window at `i` does not cross the root.

drop_psi *Lemma, line 781.* [aux]
`drop_psi` : `psi i` does not touch positions at or after the window end, so the suffix from `i + window_size i w` is unchanged.

nth_psi_left *Lemma, line 800.* [aux]
`nth_psi_left` : positionwise version of `take_psi` – entries before the window are fixed.

nth_psi_right *Lemma, line 809.* [aux]
`nth_psi_right` : positionwise version of `drop_psi` – entries after the window are fixed.

nth_psi_inside *Lemma, line 819.* [aux]
`nth_psi_inside` : within the window, `psi i w` reads off `rank_shift_seq (window_at i w)` at the offset `k - i`.

take_psi_perm *Lemma, line 844.* [aux]
`take_psi_perm` : when `j` sits past the window end, the prefix `take j (psi i w)` is a permutation of `take j w`; used in `mm_pos_psi_eq` to argue invariance of the global extremum positions.

notin_take_mm *Lemma, line 881.* [aux]
`notin_take_mm` : neither the global min nor the global max appears in the prefix `take (mm_pos w) w`, by minimality of `mm_pos`.

min_val_drop *Lemma, line 899.* [aux]
`min_val_drop` : if the global min is not in `take j w`, the min of `drop j w` equals the global min of `w`.

max_val_drop *Lemma, line 924.* [aux]
`max_val_drop` : dual of `min_val_drop` for the global max.

mm_pos_char *Lemma, line 953.* [aux]
`mm_pos_char` : the `mm_pos s` index is uniquely characterized by: no extremum sits in `take j s` and the extremum sits at `nth 0 s j`. Used to prove `mm_pos` is preserved by `psi`.

nth_w_mm_pos *Lemma, line 991.* [aux]
`nth_w_mm_pos` : the entry at `mm_pos w` is exactly the global min or the global max of `w`, by definition of `mm_pos`.

sorted_head_le *Lemma, line 1006*. [aux]
`sorted_head_le` : on a sorted sequence, the head bounds every element from below.

sorted_last_ge *Lemma, line 1019*. [aux]
`sorted_last_ge` : on a sorted sequence, the last element bounds every element from above.

min_eq_nth_sort_0 *Lemma, line 1035*. [aux]
`min_eq_nth_sort_0` : the `foldr minn` minimum equals the first entry of the sorted list – bridge from value-based to rank-based statements.

max_eq_nth_sort_last *Lemma, line 1055*. [aux]
`max_eq_nth_sort_last` : dual of `min_eq_nth_sort_0` – the `foldr maxn` maximum equals the last entry of the sorted list.

rank_shift_head_min_to_max *Lemma, line 1082*. [aux]
`rank_shift_head_min_to_max` : if the head of `L` is the minimum, the rank-shift sends it to the maximum; this is one half of the head-flip.

rank_shift_head_max_to_min *Lemma, line 1101*. [aux]
`rank_shift_head_max_to_min` : the dual head-flip – if the head of `L` is the maximum, the rank-shift sends it to the minimum.

mm_pos_psi_eq *Lemma, line 1132*. [aux]
`mm_pos_psi_eq` : `psi i` preserves the global root index `mm_pos`, by the head-flip combined with extremum-preservation away from the window. Key step toward `psi_involutive`.

ws_lt_size *Lemma, line 1259*. [aux]
`ws_lt_size` : a nontrivial window forces `i < size w`; used to discharge in-range hypotheses cheaply.

take_mm_psi *Lemma, line 1268*. [aux]
`take_mm_psi` : when `i` is in the left subtree, `psi i` commutes with `take (mm_pos w)`, i.e. acts on the left subtree's labels alone. Recursive descent step for `window_size_psi_self` and `window_at_psi_self`.

drop_mm_psi *Lemma, line 1301*. [aux]
`drop_mm_psi` : dual of `take_mm_psi` – when `i` is in the right subtree, `psi i` commutes with `drop (mm_pos w).+1` and re-indexes `i` by subtracting the left-plus-root size.

window_size_psi_self *Lemma, line 1350*. [aux]
`window_size_psi_self` : applying `psi i` to `w` does not change the window size at the same position `i`. Proved by induction on size using `take_mm_psi`/`drop_mm_psi`.

window_at_psi_self *Lemma, line 1411*. [aux]
`window_at_psi_self` : the window at `i` after one application of `psi i` is exactly the rank-shifted original window. Required by the second `psi i` application in `psi_involutive`.

window_head_extremum *Lemma, line 1489*. [aux]
`window_head_extremum` : the head of `window_at i w` is always an extremum (min or max) of the window – Stanley's structural fact (F2) transferred to the window. Discharge condition for `rank_shift_seq_involutive`.

psi_involutive *Theorem, line 1554*. [aux]
`psi_involutive` is the M2 involutivity theorem: `psi i (psi i w) = w` for every `i` and uniq `w`. Stanley fact about `psi_i`.

window_trichotomy *Lemma, line 1612.* [aux]
`window_trichotomy` : two M-windows in the same word are either disjoint (left or right) or one is nested in the other – the binary tree geometry that drives the `psi_comm` proof.

foldr_minn_le_nth *Lemma, line 1816.* [aux]
`foldr_minn_le_nth` : positionwise version of `foldr_minn_le`.

foldr_maxn_ge_nth *Lemma, line 1822.* [aux]
`foldr_maxn_ge_nth` : positionwise version of `foldr_maxn_ge`.

mm_pos_order_iso *Lemma, line 1831.* [aux]
`mm_pos_order_iso` : `mm_pos` depends only on the comparison structure of the sequence – two order-isomorphic sequences have the same `mm_pos`. Foundational for transferring tree shape across the rank-shift.

order_iso_take *Lemma, line 1937.* [aux]
`order_iso_take` : the order-isomorphism property descends to `take m` of both sequences; helper for the recursive case in `window_size_order_iso`.

order_iso_drop *Lemma, line 1955.* [aux]
`order_iso_drop` : the order-isomorphism property descends to `drop m.+1` of both sequences; companion of `order_iso_take`.

window_size_order_iso *Lemma, line 1972.* [aux]
`window_size_order_iso` : `window_size` is a function only of comparison structure – two order-isomorphic sequences have the same window sizes at every position. Lifts `mm_pos_order_iso` recursively.

mm_pos_drop_mm *Lemma, line 2039.* [aux]
`mm_pos_drop_mm` : after dropping the prefix up to the root, the new head is itself the root of its subtree, so its `mm_pos` is `0`.

psi_0_eq *Lemma, line 2071.* [aux]
`psi_0_eq` : when `mm_pos s = 0` (the head is the root), `psi 0 s` reduces to `rank_shift_seq s` – the action on the whole sequence.

drop_psi_above *Lemma, line 2096.* [aux]
`drop_psi_above` : `psi j` does not affect any suffix `drop k w` when `k` is past the window end – generalizes `drop_psi`.

psi_descent_thms.v

window_at_uniq *Lemma, line 14.* [aux]
`window_at_uniq` : uniqueness propagates from `w` to `window_at i w`.

size_window_at *Lemma, line 20.* [aux]
`size_window_at` : in-range, `size (window_at i w) = window_size i w`; the slice fits exactly.

nth_window_at *Lemma, line 31.* [aux]
`nth_window_at` : indexing into `window_at i w` is offset by `i` relative to indexing into `w`.

elem_in_range *Lemma, line 42.* [aux]
`elem_in_range` : every entry of `L` is bounded between the sorted extremes of `L`. Tag-along helper for `cmp_out_of_range`.

head_min_not_descent *Lemma, line 81.* [aux]
head_min_not_descent : if the window head is the window-min, then i is not a descent of w. Connects head-extremum to descent polarity.

head_max_is_descent *Lemma, line 104.* [aux]
head_max_is_descent : dual of head_min_not_descent – if the window head is the window-max, then i is a descent of w.

elem_rs_in_range *Lemma, line 147.* [aux]
elem_rs_in_range : every entry of rank_shift_seq L is bounded between the sorted extremes of L; rank-shift counterpart of elem_in_range.

rs_head_max_descent *Lemma, line 198.* [aux]
rs_head_max_descent : after the rank-shift sends the head to the max, position 0 becomes a descent. Used in the psi_R_add and psi_LR_swap1 cases of the descent-effect theorems.

rs_head_min_no_descent *Lemma, line 229.* [aux]
rs_head_min_no_descent : dual of rs_head_max_descent – after rank-shift sends head to min, position 0 is no longer a descent.

cmp_out_of_range *Lemma, line 261.* [aux]
cmp_out_of_range : when v lies outside the sorted range of L, the comparison nth L j > v is invariant under replacing L by rank_shift_seq L. Used at right-boundary descent positions.

cmp_out_of_range_left *Lemma, line 285.* [aux]
cmp_out_of_range_left : the cmp_out_of_range variant with v on the left of >; used at the left boundary i-1.

descent_psi_interior *Lemma, line 325.* [aux]
descent_psi_interior : at strictly interior window positions k, psi i preserves the descent value (interior order is preserved by rank-shift).

descent_psi_rboundary *Lemma, line 372.* [aux]
descent_psi_rboundary : at the right boundary k = i + ws - 1, psi i preserves the descent value, by post_window_extremum + cmp_out_of_range.

descent_psi_lboundary_R *Lemma, line 418.* [aux]
descent_psi_lboundary_R : at the left boundary i-1, for an R vertex, psi i preserves the descent value. Uses pre_window_extremum_R + cmp_out_of_range_left.

descent_psi_R_add *Lemma, line 450.* [aux]
descent_psi_R_add : R-vertex case where i is not a descent – applying psi i adds i to the descent set, leaving all other positions unchanged. Half of Stanley's Fact #2 (R case).

descent_psi_R_remove *Lemma, line 502.* [aux]
descent_psi_R_remove : R-vertex case where i is a descent – applying psi i removes i from the descent set. Other half of Stanley's Fact #2 (R case).

descent_psi_LR_swap1 *Lemma, line 554.* [aux]
descent_psi_LR_swap1 : LR-vertex case where i is not a descent (hence i-1 is) – applying psi i swaps the descent from i-1 to i. Stanley's Fact #2 (LR case, swap1).

descent_psi_LR_swap2 *Lemma, line 623.* [aux]
descent_psi_LR_swap2 : LR-vertex case where i is a descent (hence i-1 is not) – applying psi i swaps the descent from i to i-1. Stanley's Fact #2 (LR case, swap2).

psi_descent_v2.v

is_descent_seq *Definition, line 23.* [aux]
`is_descent_seq w k` is the descent predicate: `k` is a descent of `w` iff `w_k > w_{k+1}`. Sequence-level mirror of Stanley's `Des(w) = { i : w_i > w_{i+1} }`.

has_left_child_fuel *Fixpoint, line 38.* [aux]
`has_left_child_fuel fuel i s` tests, by fuel-bounded recursion on `mm_pos`, whether the vertex at in-order position `i` of `M(s)` has a left child (equivalent to it being an LR vertex).

has_left_child *Definition, line 55.* [aux]
`has_left_child i w` : the vertex at in-order position `i` of `M(w)` has a left child. Used to dispatch the descent-effect of `psi_i` into LR (with left child) and R (right-child only) cases.

has_left_child_fuel_0 *Lemma, line 60.* [aux]
`has_left_child_fuel_0` : the vertex at in-order position `0` never has a left child (it is the leftmost vertex).

has_left_child_0 *Lemma, line 69.* [aux]
`has_left_child_0` : top-level corollary of `has_left_child_fuel_0` – position `0` never has a left child.

has_left_child_fuel_monotone *Lemma, line 74.* [aux]
`has_left_child_fuel_monotone` : independence of fuel above the input size, the `has_left_child` analogue of `window_size_fuel_monotone`.

has_left_child_cons *Lemma, line 102.* [aux]
`has_left_child_cons` : structural unfolding of `has_left_child` on a `cons` by the `mm_pos` split, mirroring `window_size_cons`.

valid_mm *Fixpoint, line 149.* [aux]
`valid_mm t` : the tree `t` is a valid min-max tree, i.e. at every `Node l x r` the root sits at `mm_pos` of the in-order traversal. Specifies the trees in the image of `mmtree_of_seq_mm`.

valid_mm_build *Lemma, line 160.* [aux]
`valid_mm_build` : every output of `mmtree_of_seq_mm` satisfies `valid_mm`, the structural invariant used by `tree_structure`.

take_mm_eq *Lemma, line 214.* [aux]
`take_mm_eq` : at the `mm_pos` split, `take (size sl) s` recovers the left-subtree in-order `sl`.

drop_mm_eq *Lemma, line 219.* [aux]
`drop_mm_eq` : dual of `take_mm_eq` – `drop (size sl).+1 s` is the right-subtree in-order `sr`.

ws_bridge_left *Lemma, line 230.* [aux]
`ws_bridge_left` : on a valid node `s = sl ++ x :: sr`, when `i` is in the left subtree, `window_size i s` reduces to `window_size i sl`. Bridge between flat and structural recursion.

ws_bridge_root *Lemma, line 244.* [aux]
`ws_bridge_root` : at the root of the subtree, the M-window has size `size s - size sl` (root + right subtree).

ws_bridge_right *Lemma, line 257.* [aux]
ws_bridge_right : when i is in the right subtree, window_size i s reduces to window_size (i - size sl - 1) sr – the right-subtree descent step.

wa_bridge_left *Lemma, line 275.* [aux]
wa_bridge_left : the window_at analogue of ws_bridge_left.

wa_bridge_root *Lemma, line 289.* [aux]
wa_bridge_root : at the root, window_at is the suffix drop (size sl) s (root + right-subtree labels).

wa_bridge_right *Lemma, line 300.* [aux]
wa_bridge_right : the window_at analogue of ws_bridge_right.

hlc_bridge_left *Lemma, line 319.* [aux]
hlc_bridge_left : has_left_child structural bridge analogous to ws_bridge_left.

hlc_bridge_root *Lemma, line 333.* [aux]
hlc_bridge_root : at the root, has_left_child is true iff there is a nonempty left subtree.

hlc_bridge_right *Lemma, line 345.* [aux]
hlc_bridge_right : has_left_child structural bridge analogous to ws_bridge_right.

tree_props *Definition, line 522.* [aux]
tree_props i w is the conjunction of five structural properties at in-order position i: post-window extremum, two pre-window-vs-window inequalities, the descent flip, and the no-left-child extremum. Bundled to share a single structural induction.

pred_sub_add *Lemma, line 560.* [aux]
pred_sub_add : an arithmetic identity used to align in-order indices when descending into the right subtree.

tree_structure_via_tree *Lemma, line 581.* [aux]
tree_structure_via_tree : tree_props holds for every position i of the in-order traversal of any valid min-max tree. Proved by structural induction on t, dispatching on i vs the root index.

tree_structure *Lemma, line 1073.* [aux]
tree_structure : tree_props i w for every uniq w, obtained by re-routing through mmtree_of_seq_mm and tree_structure_via_tree.

post_window_extremum *Lemma, line 1087.* [aux]
post_window_extremum : the entry just past the window is either smaller than every window value or larger than every window value – the post-window position is a global extremum. First projection of tree_props.

pre_window_lt_max_when_min_head *Lemma, line 1116.* [aux]
pre_window_lt_max_when_min_head : at an LR vertex whose head is the window-min, the pre-window entry w_{i-1} is below the window-max. Discharges the head = min branch of exactly_one_descent_LR.

pre_window_gt_min_when_max_head *Lemma, line 1134.* [aux]
pre_window_gt_min_when_max_head : dual of pre_window_lt_max_when_min_head – at an LR vertex whose head is the window-max, w_{i-1} is above the window-min.

exactly_one_descent_LR *Lemma, line 1163.* [aux]
exactly_one_descent_LR : at an LR vertex with nontrivial window, exactly one of positions i-1, i is a descent of w. Used by the descent-effect theorems in psi_descent_thms.v.

pre_window_extremum_R *Lemma, line 1189.* [aux]
pre_window_extremum_R : at an R vertex (no left child), the entry w_{i-1} is itself a global extremum relative to the window. R-case counterpart of the LR pre-window lemmas.

qeul.v

q_eul_pol *Definition, line 41.* [Ch 6 §6.2]
q_eul_pol n is the q-Eulerian polynomial A_n(q,t) = \{}sum_(sigma in S_{n+1}) q^{}(maj sigma) * t^{}(des sigma), the bivariate joint generating function of maj and des over permutations of n+1 letters. Stanley EC1 \S{}1.4.

q_eul_pol_t1 *Theorem, line 50.* [Ch 6 §6.2]
Specialization t = 1 of A_n(q,t) recovers the q-factorial: A_n(q, 1) = [n+1]_q!. Direct consequence of maj_q_fact.

eul_pol *Definition, line 68.* [Ch 6 §6.2]
eul_pol n is the classical Eulerian polynomial \{}sum_(k <= n) eulerian(n,k) * t^{}k, with coefficients the Eulerian numbers counting permutations of n+1 letters by descent count. Stanley EC1 \S{}1.4.

sum_des_eq_eul_pol *Lemma, line 74.* [aux]
The descent generating function expressed as the Eulerian polynomial: \{}sum_(sigma in S_{n+1}) X^{}(des sigma) = eul_pol n. Obtained by partitioning permutations according to their descent count.

q1_subst *Definition, line 95.* [Ch 6 §6.2]
q1_subst is the substitution q := 1 on a bivariate polynomial in {poly {poly int}}, implemented by evaluating each inner polynomial coefficient at 1.

q_eul_pol_q1 *Theorem, line 101.* [Ch 6 §6.2]
Specialization q = 1 of A_n(q,t) recovers the classical Eulerian polynomial: A_n(1, t) = eul_pol n. Each q^(maj sigma) becomes 1, leaving the descent generating function.

qfact.v

q_int *Definition, line 28.* [Ch 6 §6.1]
q_int n is the q-integer n_q = 1 + q + q^2 + ... + q^(n-1), represented as the polynomial \{}sum_(i < n) 'X^{}i in {poly int}. Stanley EC1 \S{}1.4.

q_fact *Definition, line 32.* [Ch 6 §6.1]
q_fact n is the q-factorial n+1_q! = 1_q 2_q ... n+1_q, the product of q_int k.+1 for k < n.+1. Stanley EC1 \S{}1.4.

sum_rev_X *Lemma, line 41.* [aux]
Reindexing identity: summing X^(n - p) over p < n.+1 equals q_int n.+1. Used to recognize the q-integer factor produced by expanding exprD over val p in the inv_q_fact inductive step.

is_inv_lift_pair *Lemma, line 70.* [aux]
Inversion correspondence on the "non-p rows": a pair (lift p j1, lift p j2) is an inversion of insert_max_perm t p iff (j1, j2) is an inversion of t.

is_inv_p_lift *Lemma, line 82.* [aux]
Inversions on the "p-row": every pair `(p, lift p k)` with `p < lift p k` is an inversion of `insert_max_perm t p`, since `$\{}sigma$ p = ord_max` dominates every value `$\{}sigma$ (lift p k)`.

is_inv_lift_p *Lemma, line 93.* [aux]
Pairs `(lift p k, p)` are never inversions of `insert_max_perm t p`: `$\{}sigma$ p = ord_max` is the maximum value, so it cannot be exceeded by any earlier image `$\{}sigma$ (lift p k)`.

sum_geq_p *Lemma, line 106.* [aux]
Counts the elements `k : 'I_n.+1` with `p <= val k`: there are exactly `n.+1 - p` of them. Used to evaluate the p-row contribution in `inv_insert_max`.

inv_via_sum *Lemma, line 130.* [aux]
Reformulates `inv s` as the unconditional double sum `\{}sum_i \{}sum_j is_inv s i j`, unfolding the `##|...|` counting form.

inv_insert_max *Lemma, line 147.* [aux]
Headline arithmetic identity for the `insert_max_perm` bijection: `inv (insert_max_perm t p) = inv t + (n.+1 - val p)`. The new image `ord_max` at position `p` contributes exactly `n.+1 - val p` new inversions; the rest match `inv t` under the lift.

inv_q_fact *Theorem, line 212.* [Ch 6 §6.1]
Headline q-factorial identity for the inversion statistic (Stanley EC1 \S{}1.4, Cor 1.3.10): `\{}sum_(sigma in S_{n+1}) X^{}(inv sigma) = [n+1]_q!`. Proved by induction on `n` using the `insert_max_perm` bijection and `inv_insert_max`.

maj_q_fact *Theorem, line 247.* [Ch 6 §6.1]
Headline q-factorial identity for the major-index statistic (Stanley EC1 \S{}1.4): `\{}sum_(sigma in S_{n+1}) X^{}(maj sigma) = [n+1]_q!`. One-line transfer from `inv_q_fact` via the Foata bijection `foata_perm`, which sends `inv` to `maj`.

stirling_fiber.v

insert_after_fun *Definition, line 34.* [aux]
`insert_after_fun j0 s` is the underlying function of `insert_after j0 s`: it splices `ord_max` into the `j0`-cycle of `s` right after `j0`. See `insert_after` for the perm wrapping.

insert_after_fun_inj *Lemma, line 44.* [aux]
`insert_after_fun j0 s` is injective on `'I_n.+1`. Used to package it as a `perm` in `insert_after`.

insert_after *Definition, line 82.* [aux]
`insert_after j0 s : {perm 'I_n.+1}` is the permutation that splices `ord_max` into the `j0`-cycle of `s` right after `j0`. An `s`-cycle `p1 -> ... -> j0 -> s j0 -> ...` becomes `p1 -> ... -> j0 -> ord_max -> s j0 -> ...`.

insert_afterE *Lemma, line 86.* [aux]
`insert_after` unfolds to `insert_after_fun` pointwise.

insert_after_ord_max *Lemma, line 92.* [aux]
Defining equation: `insert_after j0 s` sends `ord_max` to `lift ord_max (s j0)`.

insert_after_j0 *Lemma, line 98.* [aux]
Defining equation: `insert_after j0 s` sends `lift ord_max j0` to `ord_max` (the spliced-in element follows `j0` in its cycle).

insert_after_other *Lemma, line 104.* [aux]
Defining equation: on points `lift ord_max k` with `k != j0`, `insert_after j0 s` acts like the `lift ord_max`-extension of `s`.

insert_after_ord_max_neq *Lemma, line 113.* [aux]
`insert_after j0 s` does not fix `ord_max`. Used to characterise the image of `insert_after` in the bijection of \S{}D.

sanity_insert_after_pointwise *Lemma, line 131.* [aux]
Symbolic sanity check: bundles the three defining equations of `insert_after j0 s` (since mathcomp's `{perm}` is opaque to `vm_compute`, we cannot do a numeric check).

insert_after_decomp *Lemma, line 160.* [aux]
Decomposition: `insert_after j0 s` factors as the transposition `tperm ord_max (lift ord_max j0)` composed with `perm.lift_perm ord_max ord_max s`. Used in `cycle_count_insert_afte` to apply `porbits_mul_tperm`.

cycle_count_insert_after *Lemma, line 180.* [aux]
Key invariant: `insert_after j0 s` preserves the cycle count of `s`. Splicing `ord_max` into the `j0`-cycle widens that cycle by one element without creating or destroying cycles.

insert_after_inj_pair *Lemma, line 212.* [aux]
`insert_after` is injective in the pair `(j0, s)`: distinct `(j0, s)` yield distinct permutations of `'I_n.+1`.

insert_after_surj *Lemma, line 239.* [aux]
Surjectivity onto the non-fixing class: every `sigma : {perm 'I_n.+1}` with `sigma ord_max != ord_max` arises as `insert_after j0 s` for some `j0` and `s`. Constructed by conjugating `sigma` with a transposition to land in the fixed-`ord_max` class, then applying `drop_perm`.

card_neq_ord_max_fiber *Lemma, line 277.* [aux]
Count of permutations `sigma` of `'I_n.+1` with `sigma ord_max != ord_max` and exactly `k.+1` cycles equals `n * stirling_c n k.+1`. This is the `j != ord_max` half of the Stirling fiber bijection.

card_eq_ord_max_fiber *Lemma, line 320.* [aux]
Count of permutations `sigma` of `'I_n.+1` fixing `ord_max` with exactly `k.+1` cycles equals `stirling_c n k`. Bijection via `perm.lift_perm ord_max ord_max` and `drop_perm`. This is the `j = ord_max` half of the Stirling fiber bijection.

stirling_c_rec *Lemma, line 375.* [Ch 2 §2.2]
Stirling cycle-number recurrence: `c(n+1, k+1) = n * c(n, k+1) + c(n, k)` (Stanley EC1 \S{}1.3.2). Final assembly from `card_neq_ord_max_fiber` and `card_eq_ord_max_fiber`.

Appendix B

Glossary of mathcomp primitives

A quick-reference companion to Chapter 1. Entries are ordered roughly by first appearance in the chapter; each gives the mathcomp source file, a one-line gloss, and an example.

`'I_n` (`fintype.v`) The finite type of natural numbers strictly less than n. Inhabitant: a pair `(i, proof of i<n)`. Cardinality n.

`ord0 : 'I_n.+1` (`fintype.v`) The value 0, with its proof of $0 < n + 1$.

`ord_max : 'I_n.+1` (`fintype.v`) The value n, the largest element.

`val i : nat` (`eqtype.v`) Strips an ordinal back to its underlying natural number.

`widen_ord (h : m \leq n) i` (`fintype.v`) Casts `i : 'I_m` to `'I_n` keeping the same numerical value. Pure cast.

`lift (p : 'I_n.+1) j` (`fintype.v`) Casts `j : 'I_n` to `'I_n.+1` "skipping" the value p. With $p =$ `ord0`, this is $j \mapsto j + 1$.

`unlift p i : option 'I_n` (`fintype.v`) Partial inverse of `lift p`. Returns `None` when $i = p$, else `Some j` with $i =$ `lift p j`.

`rev_ord i` (`fintype.v`) The involution $i \mapsto n - 1 - i$ on `'I_n`.

`{set T}` (`finset.v`) The type of finite subsets of `T` (a finite type), with extensional equality.

`[set i : T | P i]` (`finset.v`) Set comprehension: `{i : T | P i}`.

`[set f i | i in S]` (`finset.v`) Image of set S under function f.

`#|S|` (`fintype.v`) Cardinality of a finite set or predicate.

`i \in S` (`finset.v`) Membership test, returning `bool`.

`~: S` (`finset.v`) Complement of S in its ambient finite type.

`{perm T}` (`perm.v`) Type of bijections `T \to T` for `T` a finite type. Inhabitant: a function plus its proof of injectivity.

`1%g` (`fingroup.v`) Group identity. For `{perm T}`, the identity permutation.

`(s * t)%g` (`fingroup.v`) Group multiplication. For perms: `(s * t) i = t (s i)`; see lemma `permM`.

`perm (h : injective f)` (`perm.v`) Constructor: turn an injective function into a `{perm T}`.

\sum_(i : T) f i (bigop.v) Sum of `f i` over every `i : T`.

\sum_(i < n) f i (bigop.v) Same, indexed by `i : 'I_n`.

\sum_(i \in S) f i (bigop.v) Sum over the elements of an explicit set `S`.

\sum_(i | P i) f i (bigop.v) Sum over i satisfying boolean filter P. The vertical bar is a filter, not disjunction.

partition_big f xpredT (bigop.v) Rewrite a sum as a nested sum partitioned by the value of f.

n.+1, n.+2, n.-1 (ssrnat.v) Successor, double-successor, predecessor on `nat`. Read as $n+1$, $n+2$, $n-1$.

~~ b Boolean negation.

a == b (eqtype.v) Decidable equality on an `eqType`, returning `bool`. Distinct from propositional `=`; bridged by lemma `eqP`.

Section X. Variable n. ... End X. (Rocq) Sections introduce parameters that are auto-generalised on `End`. Inside the section, `n` reads as a fixed natural; outside, every definition is implicitly parameterised.

Appendix C

Source map

A complete file index is in `docs/READING_GUIDE.md` in the `mathcomp-eulerian` repository. Files covered (or planned) by this document, in Stanley reading order:

`cycles.v`, `stirling_fiber.v` Stanley §1.3.1–2 (Chapter 2).

`inversions.v` Stanley §1.3.3 (Chapter 3).

`foata.v` Stanley §1.3.4 (Chapter 4).

`descent.v`, `eulerian.v`, `beta.v` Stanley §1.4 (Chapter 5).

`qfact.v`, `qeul.v` Stanley §1.4 q-analogues (Chapter 6).

`altsub.v` Stanley §1.6.2 (Chapter 7).

`beta_omega.v`, `beta_swap.v`, `perm_seq_bridge.v` Stanley §1.6.3 (Chapter 8); the `psi_*.v` files are project-internal scaffolding for the cd-index machinery underlying `omega_proper_beta_lt`.

`experimental/reflection.v` Stanley §1.6.4 (Chapter 9); contains 1 named admit, out of the active build chain.