Swap Distance

Manuel Eberl

February 10, 2026

Given two lists that are permutations of one another, the *swap distance* (also known as the *Kendall tau distance*) is the minimum number of swap operations of adjacent elements required to make the two lists the same.

Equivalently, the swap distance of two finite linear orders $\preceq$ and $\trianglelefteq$ is the number of disagreements of the two orders, i.e. of pairs (x, y) such that $x \prec y$ and $y \lhd x$.

This article defines these two notions of swap distance as well as their equivalence under the obvious isomorphism between lists and linear orders given by interpreting a list as a *ranking* of elements in descending order.

An efficient $O(n \log n)$ algorithm to compute the swap distance is also provided via the connection to the number of inversions of a list, for which an efficient algorithm is already available in the AFP.

Contents

1 The swap distance

```
theory Swap-Distance
  imports Rankings.Rankings List-Inversions.List-Inversions
begin
```

The swap distance (also known as the Kendall tau distance) of two finite linear orders R, S is the number of pairs (x, y) such that $(x, y) \in R$ and $(y, x) \in S$.

By using the obvious correspondence between finite linear orders and lists of fixed length, the notion is transferred to lists. In this case, an alternative interpretation of the swap distance is as the smallest number of swaps of adjacent elements one can perform in order to make one list match the other one.

The swap distance is strongly related to the number of inversions of a list of linearly-ordered elements: if we rename the elements from 1 to n such that the first list becomes $[1, \ldots, n]$, the swap distance is exactly the number of inversions in the second list.

This correspondence can be used to compute the swap distance in $O(n \log n)$ time using the merge sort inversion count algorithm (which is available in the AFP).

1.1 Preliminaries

```
primrec find-index-aux :: nat ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ nat where
  find-index-aux acc P [] = acc
| find-index-aux acc P (x # xs) = (if P x then acc else find-index-aux (acc+1) P xs)

lemma find-index-aux-correct: find-index-aux acc P xs = find-index P xs + acc
  by (induction xs arbitrary: acc) simp-all

lemma find-index-aux-code [code]: find-index P xs = find-index-aux 0 P xs
  by (simp add: find-index-aux-correct)



lemma inversions-map:
  fixes xs :: 'a :: linorder list
  assumes strict-mono-on (set xs) f
  shows   inversions (map f xs) = inversions xs
proof –
  have f-less-iff: f x < f y ⟷ x < y if x ∈ set xs y ∈ set xs for x y
    using strict-mono-onD[OF assms, of x y] strict-mono-onD[OF assms, of y x] that
    by (metis not-less-iff-gr-or-eq order-less-imp-not-less)
  show ?thesis
    unfolding inversions-altdef by (auto simp: f-less-iff)
qed

lemma inversion-number-map:
  fixes xs :: 'a :: linorder list
  assumes strict-mono-on (set xs) f
  shows   inversion-number (map f xs) = inversion-number xs
```

```
  using inversions-map[OF assms] by (simp add: inversion-number-def)

lemma inversion-number-Cons:
  inversion-number (x # xs) = length (filter (λy. y < x) xs) + inversion-number xs
proof −
  have inversion-number (x # xs) = inversion-number ([x] @ xs)
    by simp
  also have ... = inversion-number xs + inversion-number-between [x] xs
    by (subst inversion-number-append) simp-all
  also have inversion-number-between [x] xs =
             card {(i, j). i = 0 ∧ j < length xs ∧ xs ! j < [x] ! i}
    by (simp add: inversion-number-between-def inversions-between-def)
  also have {(i, j). i = 0 ∧ j < length xs ∧ xs ! j < [x] ! i} =
            (λj. (0, j)) ` {j. j < length xs ∧ xs ! j < x}
    by auto
  also have card ... = card {j. j < length xs ∧ xs ! j < x}
    by (rule card-image) (auto simp: inj-on-def)
  also have ... = length (filter (λy. y < x) xs)
    by (subst length-filter-conv-card) auto
  finally show ?thesis
    by simp
qed


fun (in preorder) inversion-number-between-sorted-aux :: nat ⇒ 'a list ⇒ 'a list ⇒ nat where
  inversion-number-between-sorted-aux acc [] ys = acc
| inversion-number-between-sorted-aux acc xs [] = acc
| inversion-number-between-sorted-aux acc (x # xs) (y # ys) =
            (if ¬less y x then
               inversion-number-between-sorted-aux acc xs (y # ys)
             else
               inversion-number-between-sorted-aux (acc + length (x # xs)) (x # xs) ys)

lemma inversion-number-between-sorted-aux-correct:
  inversion-number-between-sorted-aux acc xs ys = acc + inversion-number-between-sorted xs ys
  by (induction acc xs ys rule: inversion-number-between-sorted-aux.induct) simp-all

lemma inversion-number-between-sorted-code [code]:
  inversion-number-between-sorted xs ys = inversion-number-between-sorted-aux 0 xs ys
  by (simp add: inversion-number-between-sorted-aux-correct)
```

1.2 The swap distance of two linear orders

We first define the set of "discrepancies" between the two orders.

```
definition swap-dist-relation-aux :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('a × 'a) set
where
  swap-dist-relation-aux R1 R2 = {(x,y). R1 x y ∧ ¬R1 y x ∧ R2 y x ∧ ¬R2 x y}
```

On a linear order, the following simpler definition holds.

lemma *swap-dist-relation-aux-def-linorder*:
  assumes *linorder-on A R1 linorder-on A R2*
  shows *swap-dist-relation-aux R1 R2* = {(*x*,*y*). *R1 x y* ∧ ¬*R2 x y*}
proof −
  interpret *R1*: *linorder-on A R1* **by** *fact*
  interpret *R2*: *linorder-on A R2* **by** *fact*
  show *?thesis* **unfolding** *swap-dist-relation-aux-def*
    using *R1.antisymmetric R1.total R2.antisymmetric R2.total*
      R1.refl R2.refl R1.not-outside R2.not-outside **by** *metis*
qed

lemma *swap-dist-relation-aux-same* [*simp*]: *swap-dist-relation-aux R R* = {}
  by (*auto simp*: *swap-dist-relation-aux-def*)

lemma *swap-dist-relation-aux-commute*: *swap-dist-relation-aux R1 R2* = *prod.swap* ‘ *swap-dist-relation-aux R2 R1*
  by (*auto simp*: *swap-dist-relation-aux-def*)

lemma *swap-dist-relation-aux-commute′*: *bij-betw prod.swap* (*swap-dist-relation-aux R1 R2*) (*swap-dist-relation-aux R2 R1*)
  by (*rule bij-betwI*[*of - - - prod.swap*]) (*auto simp*: *swap-dist-relation-aux-def*)

lemma *swap-dist-relation-aux-dual*:
  swap-dist-relation-aux R1 R2 = *prod.swap* ‘ *swap-dist-relation-aux* (λ*x y*. *R1 y x*) (λ*x y*. *R2 y x*)
  unfolding *swap-dist-relation-aux-def* **by** *auto*

lemma *swap-dist-relation-aux-triangle*:
  assumes *linorder-on A R1 linorder-on A R2 linorder-on A R3*
  shows *swap-dist-relation-aux R1 R3* ⊆ *swap-dist-relation-aux R1 R2* ∪ *swap-dist-relation-aux R2 R3*
proof −
  interpret *R1*: *linorder-on A R1* **by** *fact*
  interpret *R2*: *linorder-on A R2* **by** *fact*
  interpret *R3*: *linorder-on A R3* **by** *fact*
  show *?thesis*
    unfolding *swap-dist-relation-aux-def*
    using *R1.not-outside*(*1*,*2*) *R2.total R2.antisymmetric* **by** *fast*
qed

lemma *finite-swap-dist-relation-aux*:
  assumes *linorder-on A R1 finite A linorder-on B R2 finite B*
  shows *finite* (*swap-dist-relation-aux R1 R2*)
proof (*rule finite-subset*)
  interpret *R1*: *linorder-on A R1* **by** *fact*
  interpret *R2*: *linorder-on B R2* **by** *fact*
  show *swap-dist-relation-aux R1 R2* ⊆ *A* × *B*
    using *R1.not-outside R2.not-outside* **unfolding** *swap-dist-relation-aux-def* **by** *blast*

qed (*use assms* **in** *auto*)

lemma *split-Bex-pair-iff*: (∃ *z*∈*A*. *P z*) ⟷ (∃ *x y*. (*x*, *y*) ∈ *A* ∧ *P* (*x*, *y*))
by *auto*

lemma *swap-dist-relation-aux-comap-relation*:
assumes *inj-on f A linorder-on A R linorder-on A S*
shows *swap-dist-relation-aux* (*comap-relation f R*) (*comap-relation f S*) = *map-prod f f* ' *swap-dist-relation-aux R S*
(**is** *?lhs* = *?rhs*)
proof –
interpret *R*: *linorder-on A R* **by** *fact*
interpret *S*: *linorder-on A S* **by** *fact*
have (*x*, *y*) ∈ *?lhs* ⟷ (*x*, *y*) ∈ *?rhs* **for** *x y*
unfolding *swap-dist-relation-aux-def comap-relation-def map-prod-def image-iff case-prod-unfold*

split-Bex-pair-iff mem-Collect-eq fst-conv snd-conv prod.inject
using *inj-onD*[*OF assms*(*1*)] *R.not-outside S.not-outside* **by** *smt*
thus *?thesis*
by *force*
qed

lemma *swap-dist-relation-aux-restrict-subset*:
swap-dist-relation-aux (*restrict-relation A R*) (*restrict-relation A S*) ⊆
swap-dist-relation-aux R S
unfolding *swap-dist-relation-aux-def restrict-relation-def* **by** *blast*

The swap distance is then simply the number of such discrepancies.

definition *swap-dist-relation* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *nat* **where**
swap-dist-relation R1 R2 = *card* (*swap-dist-relation-aux R1 R2*)

lemma *swap-dist-relation-same* [*simp*]: *swap-dist-relation R R* = *0*
by (*simp add*: *swap-dist-relation-def*)

lemma *swap-dist-relation-commute*: *swap-dist-relation R1 R2* = *swap-dist-relation R2 R1*
using *bij-betw-same-card*[*OF swap-dist-relation-aux-commute'*[*of R1 R2*]]
by (*simp add*: *swap-dist-relation-def*)

lemma *swap-dist-relation-dual*:
swap-dist-relation R1 R2 = *swap-dist-relation* (λ*x y*. *R1 y x*) (λ*x y*. *R2 y x*)
unfolding *swap-dist-relation-def*
by (*subst swap-dist-relation-aux-dual*, *subst card-image*) *auto*

lemma *swap-dist-relation-triangle*:
assumes *linorder-on A R1 linorder-on A R2 linorder-on A R3 finite A*
shows *swap-dist-relation R1 R3* ≤ *swap-dist-relation R1 R2* + *swap-dist-relation R2 R3*
proof –
interpret *R1*: *linorder-on A R1* **by** *fact*
interpret *R2*: *linorder-on A R2* **by** *fact*

interpret *R3*: *linorder-on A R3* **by** *fact*

have *swap-dist-relation R1 R3* = *card* (*swap-dist-relation-aux R1 R3*)
by (*simp add*: *swap-dist-relation-def*)
also **{**
have *swap-dist-relation-aux R1 R3* ⊆ *swap-dist-relation-aux R1 R2* ∪ *swap-dist-relation-aux R2 R3*
by (*rule swap-dist-relation-aux-triangle*) *fact*+
moreover have *finite* (*swap-dist-relation-aux R1 R2*) *finite* (*swap-dist-relation-aux R2 R3*)
using *finite-swap-dist-relation-aux assms* **by** *blast*+
ultimately have *card* (*swap-dist-relation-aux R1 R3*) ≤ *card* (*swap-dist-relation-aux R1 R2* ∪ *swap-dist-relation-aux R2 R3*)
by (*intro card-mono*) *auto*
}
also have *card* (*swap-dist-relation-aux R1 R2* ∪ *swap-dist-relation-aux R2 R3*) ≤
card (*swap-dist-relation-aux R1 R2*) + *card* (*swap-dist-relation-aux R2 R3*)
by (*rule card-Un-le*)
also have ... = *swap-dist-relation R1 R2* + *swap-dist-relation R2 R3*
by (*simp add*: *swap-dist-relation-def*)
finally show *?thesis* **.**
qed

lemma *swap-dist-relation-aux-empty-iff*:
assumes *linorder-on A R linorder-on A S*
shows *swap-dist-relation-aux R S* = {} ⟷ *R* = *S*
proof (*rule iffI*)
fix *x y* :: *'a*
assume ∗: *swap-dist-relation-aux R S* = {}
interpret *R*: *linorder-on A R* **by** *fact*
interpret *S*: *linorder-on A S* **by** *fact*
show *R* = *S*
proof (*intro ext*)
fix *x y*
from ∗ **have** ¬*R x y* ∨ *R y x* ∨ ¬*S y x* ∨ *S x y* ¬*R y x* ∨ *R x y* ∨ ¬*S x y* ∨ *S y x*
unfolding *swap-dist-relation-aux-def* **by** *blast*+
thus *R x y* ⟷ *S x y*
using *R.total*[*of x y*] *S.total*[*of x y*] *R.not-outside*[*of x y*] *S.not-outside*[*of x y*]
R.antisymmetric[*of x y*] *S.antisymmetric*[*of x y*]
by *metis*
qed
qed *auto*

lemma *swap-dist-relation-eq-0-iff*:
assumes *linorder-on A R linorder-on A S finite A*
shows *swap-dist-relation R S* = *0* ⟷ *R* = *S*
unfolding *swap-dist-relation-def*
using *swap-dist-relation-aux-empty-iff*[*OF assms*(*1*,*2*)] *finite-swap-dist-relation-aux*[*OF assms*(*1*,*3*,*2*,*3*)]
by (*metis card-eq-0-iff*)

lemma *swap-dist-relation-comap-relation*:
  assumes *inj-on f A linorder-on A R linorder-on A S*
  shows *swap-dist-relation* (*comap-relation f R*) (*comap-relation f S*) = *swap-dist-relation R S*
proof −
  interpret *R*: *linorder-on A R* **by** *fact*
  interpret *S*: *linorder-on A S* **by** *fact*
  have *swap-dist-relation* (*comap-relation f R*) (*comap-relation f S*) = *card* (*map-prod f f* ‘ *swap-dist-relation-aux R S*)
    using *assms* **by** (*simp add*: *swap-dist-relation-def swap-dist-relation-aux-comap-relation*)
  also have ... = *swap-dist-relation R S*
    unfolding *swap-dist-relation-def*
  proof (*rule card-image*)
    show *inj-on* (*map-prod f f*) (*swap-dist-relation-aux R S*)
    proof (*rule inj-on-subset*)
      show *inj-on* (*map-prod f f*) ($A \times A$)
        using *assms*(*1*) **by** (*auto simp*: *inj-on-def*)
      show *swap-dist-relation-aux R S* $\subseteq A \times A$
        unfolding *swap-dist-relation-aux-def* **using** *R.not-outside S.not-outside* **by** *blast*
    qed
  qed
  finally show *?thesis* **.**
qed

lemma *swap-dist-relation-le*:
  assumes *preorder-on A R1 preorder-on A R2 finite A*
  shows *swap-dist-relation R1 R2* ≤ (*card A*) *choose 2*
proof −
  interpret *R1*: *preorder-on A R1* **by** *fact*
  interpret *R2*: *preorder-on A R2* **by** *fact*
  have *swap-dist-relation R1 R2* = *card* (*swap-dist-relation-aux R1 R2*)
    by (*simp add*: *swap-dist-relation-def*)
  also have *card* (*swap-dist-relation-aux R1 R2*) =
              card ((λ(*x*,*y*). {*x*,*y*}) ‘ *swap-dist-relation-aux R1 R2*)
    by (*rule card-image* [*symmetric*])
      (*auto simp*: *inj-on-def swap-dist-relation-aux-def doubleton-eq-iff*)
  also have ... ≤ *card* {*X*. *X* ⊆ *A* ∧ *card X* = *2*}
    by (*rule card-mono*)
      (*use* ‹*finite A*› *R1.not-outside R2.not-outside*
        in ‹*auto simp*: *swap-dist-relation-aux-def card-insert-if*›)
  also have ... = (*card A*) *choose 2*
    by (*rule n-subsets*) *fact*
  finally show *?thesis* **.**
qed

The swap distance reaches its maximum of $n(n-1)/2$ if and only if the two orders are inverse to each other.

lemma *swap-dist-relation-inverse*:
  assumes *linorder-on A R finite A*

```
  shows   swap-dist-relation R (λx y. R y x) = (card A) choose 2
proof −
  interpret R: linorder-on A R by fact
  have card (swap-dist-relation-aux R (λx y. R y x)) =
        card ((λ(x, y). {x, y}) ‘ swap-dist-relation-aux R (λx y. R y x))
    by (subst card-image) (auto simp: inj-on-def doubleton-eq-iff swap-dist-relation-aux-def)
  also have (λ(x, y). {x, y}) ‘ swap-dist-relation-aux R (λx y. R y x) =
            {X. X ⊆ A ∧ card X = 2}
    using R.total R.not-outside R.antisymmetric
   by (fastforce simp: swap-dist-relation-aux-def card-insert-if image-iff card-2-iff doubleton-eq-iff)
  also have card ... = (card A) choose 2
    by (rule n-subsets) fact
  finally show ?thesis
    by (simp add: swap-dist-relation-def)
qed

lemma swap-dist-relation-maximal-imp-inverse:
  assumes preorder-on A R1 preorder-on A R2 finite A
  assumes swap-dist-relation R1 R2 ≥ (card A) choose 2
  shows   R2 = (λy x. R1 x y)
proof −
  interpret R1: preorder-on A R1 by fact
  interpret R2: preorder-on A R2 by fact

  have ∗: (λ(x,y). {x,y}) ‘ swap-dist-relation-aux R1 R2 = {X. X ⊆ A ∧ card X = 2}
  proof (rule card-subset-eq)
    show finite {X. X ⊆ A ∧ card X = 2}
      using assms(3) by simp
    show (λ(x, y). {x, y}) ‘ swap-dist-relation-aux R1 R2 ⊆ {X. X ⊆ A ∧ card X = 2}
     using R1.not-outside R2.not-outside by (auto simp: swap-dist-relation-aux-def card-insert-if)
    have card ((λ(x, y). {x, y}) ‘ swap-dist-relation-aux R1 R2) = swap-dist-relation R1 R2
      unfolding swap-dist-relation-def
      by (rule card-image) (auto simp: inj-on-def swap-dist-relation-aux-def doubleton-eq-iff)
    also have ... = (card A) choose 2
      using swap-dist-relation-le[OF assms(1−3)] assms(4) by linarith
    also have ... = card {X. X ⊆ A ∧ card X = 2}
      by (rule n-subsets [symmetric]) fact
    finally show card ((λ(x, y). {x, y}) ‘ swap-dist-relation-aux R1 R2) =
                  card {X. X ⊆ A ∧ card X = 2} .
  qed

  show ?thesis
  proof (intro ext)
    fix x y :: 'a
    show R2 y x ⟷ R1 x y
    proof (cases x ∈ A ∧ y ∈ A ∧ x ≠ y)
      case False
      thus ?thesis
        using R1.refl R2.refl R1.not-outside R2.not-outside by auto
```

next
  case *True*
  hence $\{x,\ y\} \in \{X.\ X \subseteq A \wedge card\ X = 2\}$
    by *auto*
  also note $*$ [*symmetric*]
  finally show *?thesis*
    using *True* **by** (*auto simp*: *swap-dist-relation-aux-def doubleton-eq-iff*)
qed
qed
qed

lemma *swap-dist-relation-maximal-iff-inverse*:
  assumes *linorder-on A R1 linorder-on A R2 finite A*
  shows *swap-dist-relation R1 R2* = (*card A*) *choose 2* ⟷ *R2* = (λ*y x*. *R1 x y*)
proof −
  interpret *R1*: *linorder-on A R1* **by** *fact*
  interpret *R2*: *linorder-on A R2* **by** *fact*
  note *preorder* = *R1.preorder-on-axioms R2.preorder-on-axioms*
  show *?thesis*
   using *swap-dist-relation-inverse*[*OF assms*(*1*,*3*)] *swap-dist-relation-le*[*OF preorder*(*1*,*2*) *assms*(*3*)]
     swap-dist-relation-maximal-imp-inverse[*OF preorder*(*1*,*2*) *assms*(*3*)]
   by *metis*
qed

lemma *swap-dist-relation-restrict*:
  assumes *linorder-on B R linorder-on B S finite B*
  shows *swap-dist-relation* (*restrict-relation A R*) (*restrict-relation A S*) ≤
    swap-dist-relation R S
  unfolding *swap-dist-relation-def*
proof (*rule card-mono*)
  interpret *R*: *linorder-on B R* **by** *fact*
  interpret *S*: *linorder-on B S* **by** *fact*
  show *finite* (*swap-dist-relation-aux R S*)
   by (*rule finite-subset*[*of - B* × *B*])
     (*use* ‹*finite B*› *R.not-outside S.not-outside* **in** ‹*auto simp*: *swap-dist-relation-aux-def*›)
qed (*use swap-dist-relation-aux-restrict-subset*[*of A R S*] **in** *auto*)

If the restriction of two relations to some set A has the same swap distance as the full relations, the two relations must agree everywhere except inside A.

lemma *swap-dist-relation-restrict-eq-imp-eq*:
  fixes *R S A B*
  assumes *linorder-on A R linorder-on A S finite A*
  defines $R' \equiv$ *restrict-relation B R*
  defines $S' \equiv$ *restrict-relation B S*
  assumes *swap-dist-relation* $R'\ S' \geq$ *swap-dist-relation R S*
  assumes *xy*: $x \notin B \vee y \notin B$
  shows *R x y* ⟷ *S x y*
proof −

have *swap-dist-relation-aux R′ S′ = swap-dist-relation-aux R S*
proof (*rule card-subset-eq*)
show *finite* (*swap-dist-relation-aux R S*)
by (*rule finite-swap-dist-relation-aux*[*OF assms*(*1*,*3*,*2*,*3*)])
show *swap-dist-relation-aux R′ S′ ⊆ swap-dist-relation-aux R S*
unfolding *R′-def S′-def* **by** (*rule swap-dist-relation-aux-restrict-subset*)
have *swap-dist-relation R′ S′ ≤ swap-dist-relation R S*
unfolding *R′-def S′-def* **by** (*rule swap-dist-relation-restrict*[*OF assms*(*1*,*2*,*3*)])
with *assms* **have** *swap-dist-relation R′ S′ = swap-dist-relation R S*
by *linarith*
thus *card* (*swap-dist-relation-aux R′ S′*) = *card* (*swap-dist-relation-aux R S*)
by (*simp add*: *swap-dist-relation-def*)
qed
hence ∗: (*a*, *b*) ∈ *swap-dist-relation-aux R′ S′* ⟷ (*a*, *b*) ∈ *swap-dist-relation-aux R S* **for** *a b*
by *force*
interpret *R*: *linorder-on A R* **by** *fact*
interpret *S*: *linorder-on A S* **by** *fact*
show *?thesis*
using *xy* ∗[*of x y*] ∗[*of y x*] *R.not-outside*[*of x y*] *S.not-outside*[*of x y*]
R.total[*of x y*] *S.total*[*of x y*] *R.antisymmetric*[*of x y*] *S.antisymmetric*[*of x y*]
unfolding *swap-dist-relation-aux-def R′-def S′-def restrict-relation-def mem-Collect-eq prod.case*
by *metis*
qed

1.3 The swap distance of two lists

The swap distance of two lists is defined as the swap distance of the relations they correspond to when interpreting them as rankings of "biggest" to "smallest".

definition *swap-dist* :: *′a list ⇒ ′a list ⇒ nat* **where**
swap-dist xs ys =
(*if distinct xs ∧ distinct ys ∧ set xs = set ys*
then swap-dist-relation (*of-ranking xs*) (*of-ranking ys*) *else 0*)

lemma *swap-dist-le*: *swap-dist xs ys* ≤ (*length xs*) *choose 2*
proof (*cases set xs = set ys ∧ distinct xs ∧ distinct ys*)
case *True*
hence *length xs = length ys*
using *distinct-card* **by** *metis*
interpret *xs*: *linorder-on set xs of-ranking xs*
by (*rule linorder-of-ranking*) (*use True* **in** *auto*)
interpret *ys*: *linorder-on set xs of-ranking ys*
by (*rule linorder-of-ranking*) (*use True* **in** *auto*)
show *?thesis*
using *swap-dist-relation-le*[*OF xs.preorder-on-axioms ys.preorder-on-axioms*] *True*
‹*length xs = length ys*› **by** (*auto simp*: *swap-dist-def distinct-card*)
qed (*auto simp*: *swap-dist-def*)

lemma *swap-dist-same* [*simp*]: *swap-dist xs xs = 0*

by (*auto simp*: *swap-dist-def*)

lemma *swap-dist-commute*: *swap-dist xs ys* = *swap-dist ys xs*
by (*simp add*: *swap-dist-def swap-dist-relation-commute*)

lemma *swap-dist-rev* [*simp*]: *swap-dist* (*rev xs*) (*rev ys*) = *swap-dist xs ys*
proof (*cases distinct xs* ∧ *distinct ys* ∧ *set xs* = *set ys*)
case *True*
show *?thesis*
using *True swap-dist-relation-dual*[*of of-ranking xs of-ranking ys*]
by (*simp add*: *of-ranking-rev*[*abs-def*] *swap-dist-def*)
qed (*auto simp*: *swap-dist-def*)

lemma *swap-dist-rev-left*: *swap-dist* (*rev xs*) *ys* = *swap-dist xs* (*rev ys*)
using *swap-dist-rev* **by** (*metis rev-rev-ident*)

lemma *swap-dist-triangle*:
assumes *set xs* = *set ys distinct ys*
shows *swap-dist xs zs* ≤ *swap-dist xs ys* + *swap-dist ys zs*
using *swap-dist-relation-triangle*[*of set xs of-ranking xs of-ranking ys of-ranking zs*] *assms*
unfolding *swap-dist-def* **by** (*simp add*: *linorder-of-ranking*)

lemma *swap-dist-eq-0-iff*:
assumes *distinct xs distinct ys set xs* = *set ys*
shows *swap-dist xs ys* = *0* ⟷ *xs* = *ys*
proof −
have *swap-dist xs ys* = *0* ⟷ *swap-dist-relation* (*of-ranking xs*) (*of-ranking ys*) = *0*
using *assms* **by** (*auto simp*: *swap-dist-def*)
also have ... ⟷ *xs* = *ys*
using *assms* **by** (*metis List.finite-set linorder-of-ranking ranking-of-ranking swap-dist-relation-eq-0-iff*)
finally show *?thesis* **.**
qed

lemma *swap-dist-pos-iff*:
assumes *distinct xs distinct ys set xs* = *set ys*
shows *swap-dist xs ys* > *0* ⟷ *xs* ≠ *ys*
using *swap-dist-eq-0-iff*[*OF assms*] **by** *linarith*

lemma *swap-dist-map*:
assumes *inj-on f* (*set xs* ∪ *set ys*)
shows *swap-dist* (*map f xs*) (*map f ys*) = *swap-dist xs ys*
proof (*cases set xs* = *set ys* ∧ *distinct xs* ∧ *distinct ys*)
case *True*
define *A* **where** *A* = *set xs*
have *inj*: *inj-on f A*
using *assms True* **unfolding** *A-def* **by** *simp*
have *linorder*: *linorder-on A* (*of-ranking xs*) *linorder-on A* (*of-ranking ys*)
unfolding *A-def* **using** *True* **by** (*simp-all add*: *linorder-of-ranking*)
have *swap-dist* (*map f xs*) (*map f ys*) =

swap-dist-relation (*comap-relation f* (*of-ranking xs*)) (*comap-relation f* (*of-ranking ys*))
by (*use inj True* **in** ‹*auto simp*: *swap-dist-def distinct-map of-ranking-map A-def*›)
also have ... = *swap-dist xs ys*
by (*subst swap-dist-relation-comap-relation*[*OF inj linorder*])
(*use True* **in** ‹*auto simp*: *swap-dist-def*›)
finally show *?thesis* **.**
next
case *False*
have *inj*: *inj-on f* (*set xs*) *inj-on f* (*set ys*)
by (*rule inj-on-subset*[*OF assms*(*1*)]; *simp*; *fail*)+
show *?thesis* **using** *inj False inj-on-Un-image-eq-iff* [*OF assms*]
by (*auto simp*: *swap-dist-def distinct-map*)
qed

The swap distance reaches its maximum of $n(n-1)/2$ iff the two lists are reverses of one another.

lemma *swap-dist-rev-same*:
assumes *distinct xs*
shows *swap-dist xs* (*rev xs*) = (*length xs*) *choose 2*
proof −
have *swap-dist xs* (*rev xs*) = *swap-dist-relation* (*of-ranking xs*) (λ*x y*. *of-ranking xs y x*)
using *assms* **by** (*simp add*: *swap-dist-def of-ranking-rev* [*abs-def*])
also have ... = (*length xs*) *choose 2*
by (*subst swap-dist-relation-inverse*[**where** *A* = *set xs*])
(*use assms* **in** ‹*simp-all add*: *linorder-of-ranking distinct-card*›)
finally show *?thesis* **.**
qed

lemma *swap-dist-maximalD*:
assumes *set xs* = *set ys distinct xs distinct ys*
assumes *swap-dist xs ys* ≥ (*length xs*) *choose 2*
shows *ys* = *rev xs*
proof −
interpret *xs*: *linorder-on set xs of-ranking xs*
by (*rule linorder-of-ranking*) (*use assms* **in** *auto*)
interpret *ys*: *linorder-on set xs of-ranking ys*
by (*rule linorder-of-ranking*) (*use assms* **in** *auto*)
have *length xs* = *length ys*
using *assms* **by** (*metis distinct-card*)
have *of-ranking ys* = (λ*x y*. *of-ranking xs y x*)
using *assms* ‹*length xs* = *length ys*›
by (*intro swap-dist-relation-maximal-imp-inverse*[**where** *A* = *set xs*])
(*use xs.preorder-on-axioms ys.preorder-on-axioms* **in** ‹*simp-all add*: *swap-dist-def distinct-card*›)
also have ... = *of-ranking* (*rev xs*)
by (*simp add*: *fun-eq-iff*)
finally have *ranking* (*of-ranking ys*) = *ranking* (*of-ranking* (*rev xs*))
by (*rule arg-cong*)
thus *?thesis*

```
    using assms by (subst (asm) (1 2) ranking-of-ranking) auto
qed

lemma swap-dist-maximal-iff:
  assumes set xs = set ys distinct xs distinct ys
  shows   swap-dist xs ys = (length xs) choose 2 ⟷ ys = rev xs
  using assms swap-dist-maximalD[OF assms] swap-dist-le[of xs ys] swap-dist-rev-same by metis


lemma swap-dist-append-left:
  assumes distinct zs
  assumes set zs ∩ set xs = {} set zs ∩ set ys = {}
  shows   swap-dist (zs @ xs) (zs @ ys) = swap-dist xs ys
proof (cases distinct xs ∧ distinct ys ∧ set xs = set ys)
  case False
  thus ?thesis using assms
    by (auto simp: swap-dist-def)
next
  case True
  have swap-dist-relation-aux (of-ranking (zs @ xs)) (of-ranking (zs @ ys)) =
        swap-dist-relation-aux (of-ranking xs) (of-ranking ys)
    unfolding swap-dist-relation-aux-def of-ranking-append
    using assms True of-ranking-imp-in-set[of xs] of-ranking-imp-in-set[of zs]
    by blast
  thus ?thesis
    using True assms by (simp add: swap-dist-def swap-dist-relation-def)
qed

lemma swap-dist-append-right:
  assumes distinct zs
  assumes set zs ∩ set xs = {} set zs ∩ set ys = {}
  shows   swap-dist (xs @ zs) (ys @ zs) = swap-dist xs ys
proof (cases distinct xs ∧ distinct ys ∧ set xs = set ys)
  case False
  thus ?thesis using assms
    by (auto simp add: swap-dist-def Int-commute)
next
  case True
  have swap-dist-relation-aux (of-ranking (xs @ zs)) (of-ranking (ys @ zs)) =
        swap-dist-relation-aux (of-ranking xs) (of-ranking ys)
    unfolding swap-dist-relation-aux-def of-ranking-append
    using assms True of-ranking-imp-in-set[of xs] of-ranking-imp-in-set[of zs]
    by blast
  thus ?thesis
    using True assms by (simp add: swap-dist-def swap-dist-relation-def Int-commute)
qed

lemma swap-dist-Cons-same:
  assumes z ∉ set xs ∪ set ys
```

shows *swap-dist* (*z* # *xs*) (*z* # *ys*) = *swap-dist xs ys*
using *swap-dist-append-left*[*of* [*z*] *xs ys*] *assms* **by** *simp*

lemma *swap-dist-swap-first*:
assumes *distinct* (*x* # *y* # *xs*)
shows *swap-dist* (*x* # *y* # *xs*) (*y* # *x* # *xs*) = *1*
proof −
have *swap-dist* (*x* # *y* # *xs*) (*y* # *x* # *xs*) =
card (*swap-dist-relation-aux* (*of-ranking* (*x* # *y* # *xs*)) (*of-ranking* (*y* # *x* # *xs*)))
using *assms* **by** (*simp add*: *swap-dist-def swap-dist-relation-def insert-commute*)
also have *swap-dist-relation-aux* (*of-ranking* (*x* # *y* # *xs*)) (*of-ranking* (*y* # *x* # *xs*)) = {(*y*,*x*)}
using *assms of-ranking-imp-in-set*[*of xs*] **by** (*auto simp*: *swap-dist-relation-aux-def of-ranking-Cons*)
finally show *?thesis*
by *simp*
qed

1.4 The relationship between swap distance and inversions

The swap distance between a list *xs* containing the numbers $0, \ldots, n-1$ and the list $[0, \ldots, n-1]$ is exactly the number of inversions of *xs*.

lemma *swap-dist-zero-upt-n*:
assumes *mset xs* = *mset-set* {*0*..<*n*}
shows *swap-dist* [*0*..<*n*] *xs* = *inversion-number xs*
proof −
define *A* **where** *A* = {*xy*∈{..<*n*}×{..<*n*}. *fst xy* > *snd xy* ∧ *snd xy* ≺[*of-ranking xs*] *fst xy*}
define *B* **where** *B* = {*ij*∈{..<*n*}×{..<*n*}. *fst ij* < *snd ij* ∧ *xs* ! *fst ij* > *xs* ! *snd ij*}
define *f* **where** *f* = (λ*i*. *xs* ! *i*)

have *distinct*: *distinct xs*
using *assms* **by** (*metis finite-atLeastLessThan mset-eq-mset-set-imp-distinct*)
have *set-xs*: *set xs* = {*0*..<*n*}
using *assms* **by** (*metis mset-eq-setD mset-upt set-upt*)
have *length-xs*: *length xs* = *n*
using *assms* **by** (*metis diff-zero length-upt mset-eq-length mset-upt*)
have *swap-dist* ([*0*..<*n*]) *xs* = *swap-dist-relation* (*of-ranking* ([*0*..<*n*])) (*of-ranking xs*)
unfolding *swap-dist-def* **using** *distinct set-xs* **by** *simp*
also have ... = *card* (*swap-dist-relation-aux* (*of-ranking* ([*0*..<*n*])) (*of-ranking xs*))
unfolding *swap-dist-relation-def* **..**
also have *swap-dist-relation-aux* (*of-ranking* ([*0*..<*n*])) (*of-ranking xs*) = *A*
unfolding *A-def swap-dist-relation-aux-def of-ranking-zero-upt-nat strongly-preferred-def* **by** *auto*
finally have *swap-dist* ([*0*..<*n*]) *xs* = *card A* **.**

also have *bij-betw* (*map-prod f f*) *B A*
unfolding *inversions-altdef case-prod-unfold A-def B-def*
proof (*rule bij-betw-Collect*, *goal-cases*)
case *1*
have *bij-betw f* {..<*n*} (*set xs*)

```
      unfolding f-def by (rule bij-betw-nth) (use distinct in ‹simp-all add: length-xs›)
    hence bij-betw f {..<n} {..<n}
      by (simp add: set-xs atLeast0LessThan)
    show bij-betw (map-prod f f) ({..<n} × {..<n}) ({..<n} × {..<n})
      by (rule bij-betw-map-prod) fact+
  next
    case (2 xy)
    thus ?case
      using distinct
      by (auto simp: strongly-preferred-of-ranking-nth-iff f-def length-xs set-xs)
  qed
  hence card B = card A
    by (rule bij-betw-same-card)
  hence card A = card B ..
  also have card B = inversion-number xs
    unfolding inversion-number-def inversions-altdef B-def
    by (rule arg-cong[of - - card]) (auto simp: set-xs length-xs)
  finally show ?thesis .
qed
```

Hence, computing the swap distance of two arbitrary lists can be reduced to computing the number of inversions of a list by renaming all the elements such that the first list becomes $[0, \ldots, n-1]$.

```
lemma swap-dist-conv-inversion-number:
  assumes distinct: distinct xs distinct ys and set-eq: set xs = set ys
  shows   swap-dist xs ys = inversion-number (map (index xs) ys)
proof −
  have length xs = length ys
    using distinct set-eq by (metis distinct-card)
  define n where n = length xs
  have n = length ys
    using ‹length xs = length ys› unfolding n-def by simp

  define f where f = index xs
  have inj: inj-on f (set xs)
    unfolding f-def using inj-on-index[of xs] by simp

  have swap-dist xs ys = swap-dist (map f xs) (map f ys)
    by (rule swap-dist-map [symmetric]) (use set-eq inj in simp-all)
  also have map f xs = [0..<n] unfolding f-def n-def
    by (rule map-index-self) fact+
  also have swap-dist [0..<n] (map f ys) = inversion-number (map f ys)
  proof (rule swap-dist-zero-upt-n)
    show mset (map f ys) = mset-set {0..<n}
      by (metis ‹map f xs = [0..<n]› distinct(1,2) mset-map mset-set-set mset-upt set-eq)
  qed
  finally show ?thesis
    by (simp add: f-def)
qed
```

lemma *swap-dist-code'* [*code*]:
swap-dist xs ys =
(*if distinct xs* ∧ *distinct ys* ∧ *set xs* = *set ys then*
inversion-number (*map* (*index xs*) *ys*) *else 0*)
proof (*cases distinct xs* ∧ *distinct ys* ∧ *set xs* = *set ys*)
case *False*
thus *?thesis*
by (*auto simp*: *swap-dist-def*)
next
case *True*
thus *?thesis*
by (*subst swap-dist-conv-inversion-number*) *auto*
qed

1.5 Swapping adjacent list elements

definition *swap-adj-list* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
swap-adj-list i xs = (*if Suc i* < *length xs then xs*[*i* := *xs* ! *Suc i*, *Suc i* := *xs* ! *i*] *else xs*)

lemma *length-swap-adj-list* [*simp*]: *length* (*swap-adj-list i xs*) = *length xs*
by (*simp add*: *swap-adj-list-def*)

lemma *distinct-swap-adj-list-iff* [*simp*]:
distinct (*swap-adj-list i xs*) ⟷ *distinct xs*
by (*simp add*: *swap-adj-list-def*)

lemma *mset-swap-adj-list* [*simp*]:
mset (*swap-adj-list i xs*) = *mset xs*
by (*simp add*: *swap-adj-list-def mset-update*)

lemma *set-swap-adj-list* [*simp*]:
set (*swap-adj-list i xs*) = *set xs*
by (*simp add*: *swap-adj-list-def*)

lemma *swap-adj-list-append-left*:
assumes *i* ≥ *length xs*
shows *swap-adj-list i* (*xs* @ *ys*) = *xs* @ *swap-adj-list* (*i* − *length xs*) *ys*
using *assms* **by** (*auto simp*: *swap-adj-list-def list-update-append nth-append Suc-diff-le*)

lemma *swap-adj-list-Cons*:
assumes *i* > *0*
shows *swap-adj-list i* (*x* # *xs*) = *x* # *swap-adj-list* (*i* − *1*) *xs*
using *swap-adj-list-append-left*[*of* [*x*] *i xs*] *assms* **by** *simp*

lemma *swap-adj-list-append-right*:
assumes *Suc i* < *length xs*
shows *swap-adj-list i* (*xs* @ *ys*) = *swap-adj-list i xs* @ *ys*
using *assms* **by** (*auto simp*: *swap-adj-list-def list-update-append nth-append*)

```
lemma swap-dist-swap-adj-list:
  assumes Suc i < length xs distinct xs
  shows   swap-dist xs (swap-adj-list i xs) = 1
proof −
  define x y where x = xs ! i and y = xs ! Suc i
  define ys zs where ys = take i xs and zs = drop (i+2) xs
  have length ys = i
    using assms(1) by (simp add: ys-def)
  have 1: xs = ys @ x # y # zs
    unfolding x-def y-def ys-def zs-def using assms(1) by (simp add: Cons-nth-drop-Suc)
  have 2: swap-adj-list i xs = ys @ y # x # zs
    by (simp add: swap-adj-list-def 1 list-update-append ‹length ys = i› nth-append)
  have swap-dist xs (swap-adj-list i xs) =
      swap-dist (ys @ x # y # zs) (ys @ y # x # zs)
    by (subst 1, subst 2) (rule refl)
  also have ... = 1
    using assms by (simp add: 1 swap-dist-swap-first swap-dist-append-left)
  finally show ?thesis .
qed

fun swap-adjs-list :: nat list ⇒ 'a list ⇒ 'a list where
  swap-adjs-list [] xs = xs
| swap-adjs-list (i # is) xs = swap-adjs-list is (swap-adj-list i xs)

lemma length-swap-adjs-list [simp]: length (swap-adjs-list is xs) = length xs
  by (induction is arbitrary: xs) simp-all

lemma distinct-swap-adjs-list-iff [simp]:
  distinct (swap-adjs-list is xs) ⟷ distinct xs
  by (induction is arbitrary: xs) (auto simp: swap-adj-list-def)

lemma mset-swap-adjs-list [simp]:
  mset (swap-adjs-list is xs) = mset xs
  by (induction is arbitrary: xs) (auto simp: swap-adj-list-def mset-update)

lemma set-swap-adjs-list-list [simp]:
  set (swap-adjs-list is xs) = set xs
  by (induction is arbitrary: xs) (auto simp: swap-adj-list-def mset-update)

lemma swap-adjs-list-append:
  swap-adjs-list (is @ js) xs = swap-adjs-list js (swap-adjs-list is xs)
  by (induction is arbitrary: xs) simp-all

lemma swap-adjs-list-append-left:
  assumes ∀ i∈set is. i ≥ length xs
  shows   swap-adjs-list is (xs @ ys) = xs @ swap-adjs-list (map (λi. i − length xs) is) ys
  using assms by (induction is arbitrary: ys) (simp-all add: swap-adj-list-append-left)
```

lemma *swap-adjs-list-Cons*:
  assumes *0 ∉ set is*
  shows *swap-adjs-list is* (*x* # *xs*) = *x* # *swap-adjs-list* (*map* (λ*i*. *i* − *1*) *is*) *xs*
proof −
  have ∀ *i*∈*set is*. *Suc 0* ≤ *i*
    using *assms* **by** (*auto simp*: *Suc-le-eq intro*!: *Nat.gr0I*)
  thus *?thesis*
    using *swap-adjs-list-append-left*[*of is* [*x*] *xs*] **by** *simp*
qed

lemma *swap-adjs-list-append-right*:
  assumes ∀ *i*∈*set is*. *Suc i* < *length xs*
  shows *swap-adjs-list is* (*xs* @ *ys*) = *swap-adjs-list is xs* @ *ys*
  using *assms* **by** (*induction is arbitrary*: *xs*) (*simp-all add*: *swap-adj-list-append-right*)

Swapping two adjacent elements either increases or decreases the swap distance by 1, depending on the orientation of the swapped pair in the other relation.

lemma *swap-dist-relation-of-ranking-swap*:
  assumes *distinct* (*xs* @ *x* # *y* # *ys*)
  shows *swap-dist-relation R* (*of-ranking* (*xs* @ *x* # *y* # *ys*)) + (*if y* ≺[*R*] *x then 1 else 0*) =
      swap-dist-relation R (*of-ranking* (*xs* @ *y* # *x* # *ys*)) + (*if x* ≺[*R*] *y then 1 else 0*)
proof −
  have *swap-dist-relation-aux R* (*of-ranking* (*xs* @ *x* # *y* # *ys*)) ∪ (*if y* ≺[*R*] *x then* {(*y*,*x*)} *else* {}) =
      swap-dist-relation-aux R (*of-ranking* (*xs* @ *y* # *x* # *ys*)) ∪ (*if x* ≺[*R*] *y then* {(*x*,*y*)} *else* {})
    (**is** *?lhs* = *?rhs*)
    using *assms*
    by (*auto simp*: *swap-dist-relation-aux-def of-ranking-append of-ranking-Cons strongly-preferred-def*
        dest: *of-ranking-imp-in-set*)
  moreover have *card ?lhs* = *card* (*swap-dist-relation-aux R* (*of-ranking* (*xs* @ *x* # *y* # *ys*))) + (*if y* ≺[*R*] *x then 1 else 0*)
  proof (*subst card-Un-disjoint*)
    show *finite* (*swap-dist-relation-aux R* (*of-ranking* (*xs* @ *x* # *y* # *ys*)))
    proof (*rule finite-subset*)
      show *swap-dist-relation-aux R* (*of-ranking* (*xs* @ *x* # *y* # *ys*)) ⊆
          set (*xs* @ *x* # *y* # *ys*) × *set* (*xs* @ *x* # *y* # *ys*)
        unfolding *swap-dist-relation-aux-def* **using** *of-ranking-imp-in-set*[*of* (*xs* @ *x* # *y* # *ys*)]
        by *blast*
    qed *auto*
  qed (*auto simp*: *swap-dist-relation-aux-def of-ranking-append of-ranking-Cons*)
  moreover have *card ?rhs* = *card* (*swap-dist-relation-aux R* (*of-ranking* (*xs* @ *y* # *x* # *ys*))) + (*if x* ≺[*R*] *y then 1 else 0*)
  proof (*subst card-Un-disjoint*)
    show *finite* (*swap-dist-relation-aux R* (*of-ranking* (*xs* @ *y* # *x* # *ys*)))
    proof (*rule finite-subset*)
      show *swap-dist-relation-aux R* (*of-ranking* (*xs* @ *y* # *x* # *ys*)) ⊆
          set (*xs* @ *y* # *x* # *ys*) × *set* (*xs* @ *y* # *x* # *ys*)
        unfolding *swap-dist-relation-aux-def* **using** *of-ranking-imp-in-set*[*of* (*xs* @ *y* # *x* # *ys*)]

```
      by blast
    qed auto
  qed (auto simp: swap-dist-relation-aux-def of-ranking-append of-ranking-Cons)
  ultimately show ?thesis
    unfolding swap-dist-relation-def by metis
qed
```

1.6 Swapping non-adjacent list elements

If x and y are two not necessarily adjacent elements that are "in the wrong order", swapping them always strictly decreases the swap distance.

```
lemma swap-dist-relation-swap-less:
  assumes linorder-on A R finite A
  assumes xy: R x y
  assumes distinct: distinct (xs @ x # ys @ y # zs)
  assumes subset: set (xs @ x # ys @ y # zs) = A
  shows   swap-dist-relation R (of-ranking (xs @ x # ys @ y # zs)) >
          swap-dist-relation R (of-ranking (xs @ y # ys @ x # zs))
proof −
  interpret R: linorder-on A R by fact
  from distinct have [simp]: x ≠ y y ≠ x
    by auto
  have yx: ¬y ≼[R] x
    using xy R.antisymmetric[of x y] by auto

  define f where f = (λxs. swap-dist-relation-aux R (of-ranking xs))
  have fin: finite (f xs) for xs
    by (rule finite-subset[of - set xs × set xs])
       (auto simp: f-def swap-dist-relation-aux-def dest: of-ranking-imp-in-set)

  have f-eq: f xs = {(x, y). x ≺[R] y ∧ x ≻[of-ranking xs] y} for xs
    unfolding f-def swap-dist-relation-aux-def by (auto simp: strongly-preferred-def)
  have distinct xs distinct ys distinct zs
    using distinct by auto
  hence ∗: a ≺[of-ranking xs] b ⟷ a ≠ b ∧ of-ranking xs a b
          a ≺[of-ranking ys] b ⟷ a ≠ b ∧ of-ranking ys a b
          a ≺[of-ranking zs] b ⟷ a ≠ b ∧ of-ranking zs a b for a b
    by (metis linorder-of-ranking linorder-on-def order-on.antisymmetric
             strongly-preferred-def)+
  have ∗∗: a ≺[R] b ⟷ a ≠ b ∧ R a b for a b
    using R.antisymmetric R.total unfolding strongly-preferred-def by blast

  define lhs where
    lhs = f (xs @ x # ys @ y # zs) ∪ ({(y,b) |b. R y b ∧ b ∈ set ys} ∪ {(a,x) |a. R a x ∧ a ∈
set (y#ys)})
  define rhs where
    rhs = f (xs @ y # ys @ x # zs) ∪ ({(x,b) |b. R x b ∧ b ∈ set ys} ∪ {(a,y) |a. R a y ∧ a ∈
set (x#ys)})
```

```
  have lhs = rhs
  proof −
    have (a, b) ∈ lhs ⟷ (a, b) ∈ rhs for a b
    proof −
      have (a, b) ∈ lhs ⟷ (a, b) ∈ f (xs @ x # ys @ y # zs) ∨ R a b ∧ ((a = y ∧ (b = x ∨ b
∈ set ys)) ∨ (a ∈ set ys ∧ b = x))
        unfolding lhs-def using subset by auto
      also have ... ⟷ (a, b) ∈ f (xs @ y # ys @ x # zs) ∨ R a b ∧ ((a = x ∧ (b = y ∨ b ∈
set ys)) ∨ (a ∈ set ys ∧ b = y))
        using distinct subset unfolding f-eq
      by (force simp: of-ranking-strongly-preferred-Cons-iff of-ranking-strongly-preferred-append-iff

                        eq-commute not-strongly-preferred-of-ranking-iff * **)
      also have ... ⟷ (a, b) ∈ rhs
        unfolding rhs-def using subset yx by auto
      finally show (a, b) ∈ lhs ⟷ (a, b) ∈ rhs .
    qed
    thus ?thesis
      by auto
  qed

  define d1 where d1 = card {a. R a y ∧ R y a ∧ a ∈ set ys}
  define d2 where d2 = card {a. R x a ∧ R a y ∧ a ∈ set ys}

  have card lhs = card (f (xs @ x # ys @ y # zs)) +
              card ({(y,b) |b. R y b ∧ b ∈ set ys} ∪ {(a,x) |a. R a x ∧ a ∈ set (y#ys)})
    unfolding lhs-def
    by (intro card-Un-disjoint fin)
       (auto simp: f-def swap-dist-relation-aux-def of-ranking-Cons of-ranking-append
             dest: of-ranking-imp-in-set)
  also have card ({(y,b) |b. R y b ∧ b ∈ set ys} ∪ {(a,x) |a. R a x ∧ a ∈ set (y#ys)}) =
             card {(y,b) |b. R y b ∧ b ∈ set ys} + card {(a,x) |a. R a x ∧ a ∈ set (y#ys)}
    using distinct by (intro card-Un-disjoint) auto
  also have {(y,b) |b. R y b ∧ b ∈ set ys} = (λb. (y,b)) ‘ {b. R y b ∧ b ∈ set ys}
    by auto
  also have card ... = card {b. R y b ∧ b ∈ set ys}
    by (rule card-image) (auto simp: inj-on-def)
  also have {(a,x) |a. R a x ∧ a ∈ set (y#ys)} = (λa. (a,x)) ‘ {a. R a x ∧ a ∈ set (y#ys)}
    by auto
  also have card ... = card {a. R a x ∧ a ∈ set (y#ys)}
    by (rule card-image) (auto simp: inj-on-def)
  also have {a. R a x ∧ a ∈ set (y#ys)} = {a. R a x ∧ a ∈ set ys}
    using yx by auto
  finally have 1:
    card lhs =
      card (f (xs @ x # ys @ y # zs)) + card {a. R a x ∧ a ∈ set ys} + card {b. R y b ∧ b ∈
set ys}
    by (simp only: add-ac)
```

```
  have card rhs = card (f (xs @ y # ys @ x # zs)) +
            card ({(x,b) |b. R x b ∧ b ∈ set ys} ∪ {(a,y) |a. R a y ∧ a ∈ set (x#ys)})
    unfolding rhs-def
    by (intro card-Un-disjoint fin)
      (auto simp: f-def swap-dist-relation-aux-def of-ranking-Cons of-ranking-append
          dest: of-ranking-imp-in-set)
  also have card ({(x,b) |b. R x b ∧ b ∈ set ys} ∪ {(a,y) |a. R a y ∧ a ∈ set (x#ys)}) =
          card ({(x,b) |b. R x b ∧ b ∈ set ys}) + card ({(a,y) |a. R a y ∧ a ∈ set (x#ys)})
    using distinct by (intro card-Un-disjoint) auto
  also have {(x,b) |b. R x b ∧ b ∈ set ys} = (λb. (x,b)) ‘ {b. R x b ∧ b ∈ set ys}
    by auto
  also have card ... = card {b. R x b ∧ b ∈ set ys}
    by (rule card-image) (auto simp: inj-on-def)
  also have {(a,y) |a. R a y ∧ a ∈ set (x#ys)} = (λa. (a,y)) ‘ {a. R a y ∧ a ∈ set (x#ys)}
    by auto
  also have card ... = card {a. R a y ∧ a ∈ set (x#ys)}
    by (rule card-image) (auto simp: inj-on-def)
  also have {a. R a y ∧ a ∈ set (x#ys)} = {a. R a y ∧ a ∈ set ys} ∪ {x}
    using xy by auto
  also have card ... = card {a. R a y ∧ a ∈ set ys} + 1
    using distinct by (subst card-Un-disjoint) auto

  finally have 2:
    card rhs =
      card (f (xs @ y # ys @ x # zs)) + card {a. R a y ∧ a ∈ set ys} + card {b. R x b ∧ b ∈
set ys} + 1
    by (simp only: add-ac)

  have 3: card {a. R a x ∧ a ∈ set ys} ≤ card {a. R a y ∧ a ∈ set ys}
    by (rule card-mono) (use xy R.trans in auto)
  have 4: card {b. R y b ∧ b ∈ set ys} ≤ card {b. R x b ∧ b ∈ set ys}
    by (rule card-mono) (use xy R.trans in auto)

  have int (card lhs) = int (card rhs)
    using ‹lhs = rhs› by (rule arg-cong)
  hence int (card lhs) − card {a. R a x ∧ a ∈ set ys} − card {b. R y b ∧ b ∈ set ys} ≥
        int (card rhs) − card {a. R a y ∧ a ∈ set ys} − card {b. R x b ∧ b ∈ set ys}
    using 3 4 by linarith
  hence card (f (xs @ x # ys @ y # zs)) > card (f (xs @ y # ys @ x # zs))
    unfolding 1 2 by simp
  thus ?thesis
    unfolding f-def swap-dist-relation-def by simp
qed

lemma swap-dist-relation-swap-less':
  assumes xy: R (ys ! i) (ys ! j) ⟷ i < j
  assumes R: finite-linorder-on A R
  assumes distinct: distinct ys set ys = A
  assumes ij: i < length ys j < length ys i ≠ j
```

```
  shows   swap-dist-relation R (of-ranking ys) >
          swap-dist-relation R (of-ranking (ys[i := ys ! j, j := ys ! i]))
  using ij xy
proof (induction i j rule: linorder-wlog)
  case (le i j)
  hence i < j
    by linarith
  interpret R: finite-linorder-on A R
    by fact
  define ys1 ys2 ys3 where ys1 = take i ys
    and ys2 = take (j − i − 1) (drop (i+1) ys) and ys3 = drop (j+1) ys
  have [simp]: length ys1 = i length ys2 = j − i − 1 length ys3 = length ys − j − 1
    using le by (simp-all add: ys1-def ys2-def ys3-def)
  define y y' where y = ys ! i and y' = ys ! j

  have ys-eq: ys = ys1 @ y # ys2 @ y' # ys3
    apply (subst id-take-nth-drop[of i])
    subgoal by (use le in simp)
    apply (subst id-take-nth-drop[of j − i − 1 drop (Suc i) ys])
     apply (use le in ‹simp-all add: ys1-def ys2-def ys3-def y-def y'-def›)
    done

  have swap-dist-relation R (of-ranking (ys1 @ y # ys2 @ y' # ys3)) >
        swap-dist-relation R (of-ranking (ys1 @ y' # ys2 @ y # ys3))
  proof (rule swap-dist-relation-swap-less)
    show linorder-on A R ..
    show R y y'
      unfolding y-def y'-def using le by auto
    show distinct (ys1 @ y # ys2 @ y' # ys3)
      using distinct unfolding ys-eq by simp
    show set (ys1 @ y # ys2 @ y' # ys3) = A
      using distinct unfolding ys-eq by simp
  qed auto
  also have ys1 @ y # ys2 @ y' # ys3 = ys
    using ys-eq by simp
  also have ys1 @ y' # ys2 @ y # ys3 = ys[i := y', j := y]
    by (subst ys-eq) (use le ‹i < j› in ‹auto simp: list-update-append split: nat.splits›)
  finally show ?case
    unfolding swap-dist-def y-def y'-def using distinct by simp
next
  case (sym i j)
  interpret R: finite-linorder-on A R
    by fact
  have swap-dist-relation R (of-ranking (ys[j := ys ! i, i := ys ! j])) <
        swap-dist-relation R (of-ranking ys)
  proof (rule sym.IH)
    show R (ys ! j) (ys ! i) ⟷ (j < i)
      using sym.prems distinct R.antisymmetric R.total'
      by (metis less-imp-le-nat linorder-not-le nat-neq-iff nth-eq-iff-index-eq nth-mem)
```

qed (*use sym.prems* **in** *auto*)
thus *?case*
using *sym.prems* **by** (*simp add*: *list-update-swap*)
qed

The following formulation for lists is probably the nicest one.

lemma *swap-dist-swap-less*:
assumes *xy*: *of-ranking xs* (*ys* ! *i*) (*ys* ! *j*) ⟷ *i* < *j*
assumes *distinct*: *distinct xs distinct ys set xs* = *set ys*
assumes *ij*: *i* < *length ys j* < *length ys i* ≠ *j*
shows *swap-dist xs ys* > *swap-dist xs* (*ys*[*i* := *ys* ! *j*, *j* := *ys* ! *i*])
proof −
have *swap-dist-relation* (*of-ranking xs*) (*of-ranking ys*) >
swap-dist-relation (*of-ranking xs*) (*of-ranking* (*ys*[*i* := *ys* ! *j*, *j* := *ys* ! *i*]))
by (*rule swap-dist-relation-swap-less′*[**where** *A* = *set xs*])
(*use assms* **in** ‹*auto intro*: *finite-linorder-of-ranking*›)
thus *?thesis*
using *distinct* **by** (*simp add*: *swap-dist-def*)
qed

1.7 Swap distance as minimal number of adjacent swaps to make two lists equal

The swap distance between the original list and the list obtained after swapping adjacent elements n times is at most n.

lemma *swap-dist-swap-adjs-list*:
assumes *distinct xs*
shows *swap-dist xs* (*swap-adjs-list is xs*) ≤ *length is*
using *assms*
proof (*induction is arbitrary*: *xs*)
case (*Cons i is xs*)
define *ys* **where** *ys* = *swap-adj-list i xs*
have *swap-dist xs* (*swap-adjs-list* (*i*#*is*) *xs*) =
swap-dist xs (*swap-adjs-list is ys*)
by (*simp add*: *ys-def*)
also have ... ≤ *swap-dist xs ys* + *swap-dist ys* (*swap-adjs-list is ys*)
by (*rule swap-dist-triangle*) (*use Cons.prems* **in** ‹*simp-all add*: *ys-def*›)
also have *swap-dist xs ys* ≤ *1*
proof (*cases Suc i* < *length xs*)
case *True*
hence *swap-dist xs ys* = *1*
unfolding *ys-def* **by** (*intro swap-dist-swap-adj-list*) (*use Cons.prems* **in** *auto*)
thus *?thesis*
by *simp*
qed (*auto simp*: *ys-def swap-adj-list-def*)
also have *swap-dist ys* (*swap-adjs-list is ys*) ≤ *length is*
by (*rule Cons.IH*) (*use Cons.prems* **in** ‹*auto simp*: *ys-def*›)
finally show *?case*

  by *simp*
qed *simp-all*

Phrased in another way, any sequence of adjacent swaps that makes two lists the same must have a length at least as big as the swap distance of the two lists.

theorem *swap-dist-minimal*:
  assumes *distinct xs*
  assumes $\forall$ *i*$\in$*set is. Suc i* $<$ *length xs*
  assumes *swap-adjs-list is xs* $=$ *ys*
  shows *length is* $\geq$ *swap-dist xs ys*
  using *swap-dist-swap-adjs-list*[*of xs is*] *assms* **by** *blast*

Next, we will show that this lower bound is sharp, i.e. there exists a sequence of swaps that makes the two lists the same whose length is exactly the swap distance.

To this end, we derive an algorithm to compute a sequence of swaps whose effect is equivalent to the permutation $[0, 1, \ldots, n-1] \mapsto [i_0, i_1, \ldots, i_{n-1}]$.

We first define the following function, which returns a list of swaps that pulls the i-th element of a list to the front, i.e. it corresponds to the permutation $[0, 1, \ldots, n-1] \mapsto [i, 0, 1, \ldots, i-1, i+1, \ldots, n-1]$.

definition *pull-to-front-swaps* :: *nat* $\Rightarrow$ *nat list* **where**
  pull-to-front-swaps i $=$ *rev* [*0..*$<$*i*]

lemma *length-pull-to-front-swaps* [*simp*]: *length* (*pull-to-front-swaps i*) $=$ *i*
  by (*simp add*: *pull-to-front-swaps-def*)

lemma *set-pull-to-front-swaps* [*simp*]: *set* (*pull-to-front-swaps i*) $=$ {*0..*$<$*i*}
  by (*simp add*: *pull-to-front-swaps-def*)

lemma *pull-to-front-swaps-0* [*simp*]: *pull-to-front-swaps 0* $=$ []
  and *pull-to-front-swaps-Suc*: *pull-to-front-swaps* (*Suc i*) $=$ *i* # *pull-to-front-swaps i*
  by (*simp-all add*: *pull-to-front-swaps-def*)

lemma *swap-adjs-list-pull-to-front*:
  assumes *i* $<$ *length xs*
  shows *swap-adjs-list* (*pull-to-front-swaps i*) *xs* $=$ (*xs* ! *i*) # *take i xs* @ *drop* (*Suc i*) *xs*
  using *assms*
proof (*induction i arbitrary*: *xs*)
  case *0*
  have *xs* $=$ *xs* ! *0* # *drop* (*Suc 0*) *xs*
    using *0* **by** (*cases xs*) *auto*
  thus *?case* **by** *simp*
next
  case (*Suc i xs*)
  define *x y* **where** *x* $=$ *xs* ! *i* **and** *y* $=$ *xs* ! *Suc i*
  define *ys zs* **where** *ys* $=$ *take i xs* **and** *zs* $=$ *drop* (*i+2*) *xs*
  have [*simp*]: *length ys* $=$ *i*
    using *Suc.prems* **by** (*simp add*: *ys-def*)
  have *xs-eq*: *xs* $=$ *ys* @ *x* # *y* # *zs*

unfolding *x-def y-def ys-def zs-def* **using** *Suc.prems* **by** (*simp add*: *Cons-nth-drop-Suc*)

have *swap-adjs-list* (*pull-to-front-swaps* (*Suc i*)) *xs* =
  y # *ys* @ *drop* (*Suc i*) (*xs*[*i* := *y*, *Suc i* := *x*]) **using** *Suc.prems*
  by (*simp add*: *pull-to-front-swaps-Suc swap-adj-list-def Suc.IH x-def y-def ys-def zs-def*)
also have *drop* (*Suc i*) (*xs*[*i* := *y*, *Suc i* := *x*]) = *x* # *zs*
  by (*simp add*: *xs-eq list-update-append*)
also have *y* # *ys* @ *x* # *zs* = *xs* ! *Suc i* # *take* (*Suc i*) *xs* @ *drop* (*Suc* (*Suc i*)) *xs*
  by (*simp add*: *xs-eq nth-append*)
finally show *?case* **.**
qed

We now simply perform the "pull to front" operation so that the first element is the desired one. We then do the same thing again for the remaining $n - 1$ indices (shifted accordingly) etc. until we reach the end of the index list.

This corresponds to a variant of selection sort that only uses adjacent swaps, or it can also be seen as a kind of reversal of insertion sort.

fun *swaps-of-perm* :: *nat list* ⇒ *nat list* **where**
  swaps-of-perm [] = []
| *swaps-of-perm* (*i* # *is*) =
  pull-to-front-swaps i @ *map Suc* (*swaps-of-perm* (*map* (λ*j*. *if j* ≥ *i then j* − *1 else j*) *is*))

lemma *set-swaps-of-perm-subset*: *set* (*swaps-of-perm is*) ⊆ (⋃*i*∈*set is*. {*0*..<*i*})
  by (*induction is rule*: *swaps-of-perm.induct*; *fastforce*)

lemma *swap-adjs-list-swaps-of-perm-aux*:
  fixes *i* :: *nat*
  assumes *mset* (*i* # *is*) = *mset-set* {*0*..<*n*}
  shows *mset* (*map* (λ*j*. *if i* ≤ *j then j* − *1 else j*) *is*) = *mset-set* {*0*..<*n* − *1*}
proof −
  define *is1* **where** *is1* = *filter-mset* (λ*j*. *i* ≤ *j*) (*mset is*)
  define *is2* **where** *is2* = *filter-mset* (λ*j*. ¬(*i* ≤ *j*)) (*mset is*)

  have *i* ∈# *mset* (*i* # *is*)
    by *simp*
  also have *mset* (*i* # *is*) = *mset-set* {*0*..<*n*}
    by *fact*
  finally have *i*: *i* < *n*
    by *simp*

  have *mset-set* {*0*..<*n*} = *mset* (*i* # *is*)
    using *assms* **by** *simp*
  also have . . . = *add-mset i* (*mset is*)
    by *simp*
  finally have *mset is* = *mset-set* {*0*..<*n*} − {#*i*#}
    by *simp*
  also have . . . = *mset-set* ({*0*..<*n*} − {*i*})
    by (*subst mset-set-Diff*) (*use i* **in** *auto*)
  finally have *mset-is*: *mset is* = *mset-set* ({*0*..<*n*} − {*i*}) **.**

have *mset* (*map* (λ*j*. *if i* ≤ *j then j* − *1 else j*) *is*) =
{#*if i* ≤ *j then j* − *1 else j*. *j* ∈# *mset is*#}
by *simp*
also have *mset is* = *is1* + *is2*
unfolding *is1-def is2-def* **by** (*rule multiset-partition*)
also have {#*if i* ≤ *j then j* − *1 else j*. *j* ∈# *is1* + *is2*#} =
{#*j* − *1*. *j* ∈# *is1*#} + {#*j*. *j* ∈# *is2*#} **unfolding** *image-mset-union*
by (*intro arg-cong2*[*of* - - - - (+)] *image-mset-cong*) (*auto simp*: *is1-def is2-def*)
also have {#*j* − *1*. *j* ∈# *is1*#} = {#*j* − *1* . *j* ∈# *mset-set* {*x*. *x* < *n* ∧ *x* ≠ *i* ∧ *i* ≤ *x*}#}
unfolding *is1-def* **by** (*simp add*: *mset-is*)
also have ... = *mset-set* ((λ*j*. *j* − *1*) ‘ {*x*. *x* < *n* ∧ *x* ≠ *i* ∧ *i* ≤ *x*})
by (*intro image-mset-mset-set*) (*auto simp*: *inj-on-def*)
also have {*x*. *x* < *n* ∧ *x* ≠ *i* ∧ *i* ≤ *x*} = {*i*<..<*n*}
by *auto*
also have *bij-betw* (λ*j*. *j* − *1*) {*i*<..<*n*} {*i*..<*n* − *1*}
by (*rule bij-betwI*[*of* - - - λ*i*. *i*+*1*]) *auto*
hence (λ*j*. *j* − *1*) ‘ {*i*<..<*n*} = {*i*..<*n* − *1*}
by (*simp add*: *bij-betw-def*)
also have {#*j*. *j* ∈# *is2*#} = *mset-set* {*x*. *x* < *n* ∧ *x* ≠ *i* ∧ ¬ *i* ≤ *x*}
by (*simp add*: *is2-def mset-is*)
also have {*x*. *x* < *n* ∧ *x* ≠ *i* ∧ ¬ *i* ≤ *x*} = {..<*i*}
using *i* **by** *auto*
also have *mset-set* {*i*..<*n* − *1*} + *mset-set* {..<*i*} =
mset-set ({*i*..<*n* − *1*} ∪ {..<*i*})
by (*rule mset-set-Union* [*symmetric*]) *auto*
also have {*i*..<*n* − *1*} ∪ {..<*i*} = {*0*..<*n* − *1*}
using *i* **by** *auto*
finally show *?thesis* **.**
qed

The following result shows that the list of swaps returned by *swaps-of-perm* indeed have the desired effect.

lemma *swap-adjs-list-swaps-of-perm*:
assumes *mset is* = *mset-set* {*0*..<*length xs*}
shows *swap-adjs-list* (*swaps-of-perm is*) *xs* = *map* (λ*i*. *xs* ! *i*) *is*
using *assms*
proof (*induction is arbitrary*: *xs rule*: *swaps-of-perm.induct*)
case (*1 xs*)
thus *?case*
by (*simp add*: *mset-set-empty-iff*)
next
case (*2 i is xs*)
define *is*′ **where** *is*′ = *map* (λ*j*. *if i* ≤ *j then j* − *1 else j*) *is*
have *i*: *i* < *length xs*
proof −
have *i* ∈# *mset* (*i* # *is*)
by *simp*
also have *mset* (*i* # *is*) = *mset-set* {*0*..<*length xs*}

```
      by fact
    finally show ?thesis
      by simp
  qed
  have distinct (i # is)
    using 2.prems by (metis distinct-upt mset-eq-imp-distinct-iff mset-upt)

  have swap-adjs-list (swaps-of-perm (i # is)) xs =
          swap-adjs-list (map Suc (swaps-of-perm is′))
            (swap-adjs-list (pull-to-front-swaps i) xs)
    by (simp add: swap-adjs-list-append is′-def)
  also have swap-adjs-list (pull-to-front-swaps i) xs = xs ! i # take i xs @ drop (Suc i) xs
    by (subst swap-adjs-list-pull-to-front) (use i in auto)
  also have swap-adjs-list (map Suc (swaps-of-perm is′)) ... =
              xs ! i # swap-adjs-list (swaps-of-perm is′) (take i xs @ drop (Suc i) xs)
    by (subst swap-adjs-list-Cons) (simp-all add: o-def)
  also have swap-adjs-list (swaps-of-perm is′) (take i xs @ drop (Suc i) xs) =
                map ((!) (take i xs @ drop (Suc i) xs)) is′
    unfolding is′-def
  proof (rule 2.IH)
    have mset (map (λj. if i ≤ j then j − 1 else j) is) =
            mset-set {0..<length xs − 1}
      by (rule swap-adjs-list-swaps-of-perm-aux) (use 2.prems in simp-all)
    also have length xs − 1 = length (take i xs @ drop (Suc i) xs)
      using i by simp
    finally show mset (map (λj. if i ≤ j then j − 1 else j) is) =
                    mset-set {0..<length (take i xs @ drop (Suc i) xs)} .
  qed
  also have xs ! i # map ((!) (take i xs @ drop (Suc i) xs)) is′ = map ((!) xs) (i # is)
    by (rule nth-equalityI)
       (use i ‹distinct (i # is)›
        in ‹force simp: is′-def o-def nth-append nth-Cons set-conv-nth split: nat.splits›)+
  finally show ?case .
qed
```

The number of swaps returned by *swaps-of-perm* is exactly the number of inversions in the input list (i.e. of the index permutation described by it).

```
lemma length-swaps-of-perm:
  assumes mset is = mset-set {0..<length is}
  shows   length (swaps-of-perm is) = inversion-number is
  using assms
proof (induction is rule: swaps-of-perm.induct)
  case (2 i is)
  define n where n = length is
  define is′ where is′ = map (λj. if i ≤ j then j − 1 else j) is
  have mset is′ = mset-set {0..<Suc (length is)−1}
    unfolding is′-def by (rule swap-adjs-list-swaps-of-perm-aux) (use 2.prems in simp-all)
  also have Suc (length is) − 1 = length (map (λj. if i ≤ j then j − 1 else j) is)
    by simp
```

finally have *is*′: *mset is*′ = *mset-set* {*0*..<...} **.**

have *i*: $i \leq n$
proof −
  have *i* ∈# *mset* (*i* # *is*)
    by *simp*
  also have *mset* (*i* # *is*) = *mset-set* {*0*..*n*}
    unfolding *n-def* **using** *2.prems* **by** (*simp add*: *atLeastLessThanSuc-atLeastAtMost*)
  finally show *?thesis*
    by *simp*
qed

have *mset-set* {*0*..*n*} = *mset* (*i* # *is*)
  using *2.prems* **by** (*simp add*: *n-def atLeastLessThanSuc-atLeastAtMost*)
also have ... = *add-mset i* (*mset is*)
  by *simp*
finally have *mset is* = *mset-set* {*0*..*n*} − {#*i*#}
  by *simp*
also have ... = *mset-set* ({*0*..*n*} − {*i*})
  by (*subst mset-set-Diff*) (*use i* **in** *auto*)
finally have *mset-is*: *mset is* = *mset-set* ({*0*..*n*} − {*i*}) **.**

have *set-is*: *set is* = {*0*..*n*} − {*i*}
proof −
  have *set is* = *set-mset* (*mset is*)
    by *simp*
  also have ... = {*0*..*n*} − {*i*}
    by (*subst mset-is*) *simp-all*
  finally show *?thesis* **.**
qed

have *length* (*swaps-of-perm* (*i* # *is*)) = *i* + *length* (*swaps-of-perm is*′)
  by (*simp add*: *is*′*-def*)
also have *length* (*swaps-of-perm is*′) = *inversion-number is*′
  using *is*′ **unfolding** *is*′*-def* **by** (*rule 2.IH*)
also have *inversion-number is*′ = *inversion-number is* **unfolding** *is*′*-def*
  by (*rule inversion-number-map*) (*auto intro*!: *strict-mono-onI simp*: *set-is split*: *if-splits*)
finally have *1*: *length* (*swaps-of-perm* (*i* # *is*)) = *i* + *inversion-number is*
  by *simp*

have *inversion-number* (*i* # *is*) = *length* (*filter* (λ*y*. $y < i$) *is*) + *inversion-number is*
  by (*simp add*: *is*′*-def inversion-number-Cons*)
also have *length* (*filter* (λ*y*. $y < i$) *is*) = *size* (*filter-mset* (λ*y*. $y < i$) (*mset is*))
  by (*metis mset-filter size-mset*)
also have ... = *card* {*x*. $x \leq n \wedge x \neq i \wedge x < i$}
  by (*subst mset-is*) *simp*
also have {*x*. $x \leq n \wedge x \neq i \wedge x < i$} = {*0*..<*i*}
  using *i* **by** *auto*
also have *card* ... = *i*

```
    by simp
  finally have 2: inversion-number (i # is) = i + inversion-number is .

  show ?case
    using 1 2 by metis
qed simp-all
```

Finally, we use the above to give a list of swap operations that map one list to another. The number of swap operations produced by this is exactly the swap distance of the two lists.

```
definition swaps-of-perm' :: 'a list ⇒ 'a list ⇒ nat list where
  swaps-of-perm' xs ys = swaps-of-perm (map (index xs) ys)

theorem swaps-of-perm':
  assumes distinct xs distinct ys set xs = set ys
  shows   ∀ i∈set (swaps-of-perm' xs ys). Suc i < length xs
          swap-adjs-list (swaps-of-perm' xs ys) xs = ys
          length (swaps-of-perm' xs ys) = swap-dist xs ys
proof −
  have length-eq: length xs = length ys
    using assms by (metis distinct-card)
  have mset-eq: mset xs = mset ys
    using assms by (simp add: set-eq-iff-mset-eq-distinct)

  have mset-eq': image-mset (index xs) (mset ys) = mset-set {0..<length xs}
    by (metis assms(1) map-index-self mset-eq mset-map mset-upt)

  have swap-adjs-list (swaps-of-perm' xs ys) xs = map ((!) xs) (map (index xs) ys)
    unfolding swaps-of-perm'-def
    by (rule swap-adjs-list-swaps-of-perm) (simp add: mset-eq')
  also have ... = map id ys
    unfolding map-map by (intro map-cong) (simp-all add: assms)
  finally show swap-adjs-list (swaps-of-perm' xs ys) xs = ys
    by simp

  have set (swaps-of-perm' xs ys) ⊆ (⋃ i∈set (map (index xs) ys). {0..<i})
    unfolding swaps-of-perm'-def by (rule set-swaps-of-perm-subset)
  also have set (map (index xs) ys) = {0..<length xs}
    by (simp add: assms(1,3) index-image)
  also have (⋃ i∈{0..<length xs}. {0..<i}) ⊆ {i. Suc i < length xs}
    by auto
  finally show ∀ i∈set (swaps-of-perm' xs ys). Suc i < length xs
    by blast

  have length (swaps-of-perm' xs ys) = inversion-number (map (index xs) ys)
   unfolding swaps-of-perm'-def by (rule length-swaps-of-perm) (simp-all add: mset-eq' length-eq)
  also have ... = swap-dist xs ys
    using assms by (simp add: swap-dist-conv-inversion-number)
  finally show length (swaps-of-perm' xs ys) = swap-dist xs ys .
```

qed

Finally, we can derive the alternative characterisation of the swap distance.

lemma *swap-dist-altdef*:
  assumes *distinct xs distinct ys set xs* = *set ys*
  shows *swap-dist xs ys* = (*INF is*∈{*is. swap-adjs-list is xs* = *ys*}. *length is*)
proof (*rule antisym*)
  show *swap-dist xs ys* ≤ (*INF is*∈{*is. swap-adjs-list is xs* = *ys*}. *length is*)
  proof (*rule cINF-greatest*)
    show {*is. swap-adjs-list is xs* = *ys*} ≠ {}
      using *swaps-of-perm'*[*OF assms*] **by** *auto*
    show *swap-dist xs ys* ≤ *length is* **if** *is* ∈ {*is. swap-adjs-list is xs* = *ys*} **for** *is*
      using *that assms*(*1*) *swap-dist-swap-adjs-list* **by** *auto*
  qed
next
  have (*INF is*∈{*is. swap-adjs-list is xs* = *ys*}. *length is*) ≤ *length* (*swaps-of-perm' xs ys*)
    by (*rule cINF-lower*) (*use swaps-of-perm'*[*OF assms*] **in** *auto*)
  also have . . . = *swap-dist xs ys*
    using *swaps-of-perm'*[*OF assms*] **by** *simp*
  finally show *swap-dist xs ys* ≥ (*INF is*∈{*is. swap-adjs-list is xs* = *ys*}. *length is*) **.**
qed

end

References

[1] A. Belov and J. Marques-Silva. Muser2: An efficient MUS extractor. *J. Satisf. Boolean Model. Comput.*, 8(3/4):123–128, 2012.

[2] A. Biere, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Report Series B*, pages 10–13. University of Helsinki, 2021.

[3] P. Lammich. The GRAT tool chain – efficient (UN)SAT certificate checking with formal correctness guarantees. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 457–463. Springer, 2017.

[4] N. Wetzler, M. Heule, and W. A. H. Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014, Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.