The Inversions of a List

Manuel Eberl

June 11, 2019

Abstract

This entry defines the set of inversions of a list, i.e. the pairs of indices that violate sortedness. It also proves the correctness of the well-known $O(n \log n)$ divide-and-conquer algorithm to compute the number of inversions.

Contents

1 The Inversions of a List
  1.1 Definition of inversions .............................. 1
  1.2 Counting inversions ................................. 3
  1.3 Stability of inversions between lists under permutations . . . 5
  1.4 Inversions between sorted lists ...................... 6
  1.5 Merge sort ........................................ 8
  1.6 Merge sort with inversion counting ................ 9

1 The Inversions of a List

theory List-Inversions
  imports Main HOL-Library.Permutations
begin

1.1 Definition of inversions

context preorder
begin

We define inversions as pair of indices w.r.t. a preorder.

inductive-set inversions :: 'a list ⇒ (nat × nat) set for xs :: 'a list where
  i < j ⇒ j < length xs ⇒ less (xs ! j) (xs ! i) ⇒ (i, j) ∈ inversions xs

lemma inversions-subset: inversions xs ⊆ Sigma {..<length xs} (λi. {i<..<length xs})
  by (auto simp: inversions.simps)
lemma finite-inversions [intro]: finite (inversions xs)
by (rule finite-subset[OF inversions-subset]) auto

lemma inversions-altdef: inversions xs = \{(i, j). i < j ∧ j < length xs ∧ less (xs ! j) (xs ! i)\}
by (auto simp: inversions, simps)

lemma inversions-code:
    inversions xs = Σ {..<length xs} (λi. Set.filter (λj. less (xs ! j) (xs ! i)) {i<..<length xs})
by (auto simp: inversions-altdef)

lemmas (in -) [code] = inversions-code

lemma inversions-trivial [simp]: length xs ≤ Suc 0 ⇒ inversions xs = {}
by (auto simp: inversions-altdef)

lemma inversions-imp-less:
    z ∈ inversions xs ⇒ fst z < snd z
z ∈ inversions xs ⇒ snd z < length xs
by (auto simp: inversions-altdef)

lemma inversions-Nil [simp]: inversions [] = {}
by (auto simp: inversions-altdef)

lemma inversions-Cons:
    inversions (x # xs) =
        (λj. (0, j + 1)) • (j∈{..<length xs}. less (xs ! j) x) ∪
        map-prod Suc Suc • inversions xs (is - = ?rhs)
proof -
    have z ∈ inversions (x # xs) ↔ z ∈ ?rhs for z
    by (cases z) (auto simp: inversions-altdef map-prod-def nth-Cons split: nat.splits)
    thus ?thesis by blast
qed

The following function returns the inversions between two lists, i.e. all pairs of an element in the first list with an element in the second list such that the former is greater than the latter.

definition inversions-between :: 'a list ⇒ 'a list ⇒ (nat × nat) set
where
    inversions-between xs ys =
        \{(i, j). i < length xs ∧ j < length ys ∧ less (ys ! j) (xs ! i)\}

lemma finite-inversions-between [intro]: finite (inversions-between xs ys)
by (rule finite-subset[OF \cdot \cdot<length xs} × \cdot \cdot<length xs + length ys\}])
    (auto simp: inversions-between-def)

lemma inversions-between-Nil [simp]:
    inversions-between [] ys = {}

2
inversions-between \( \text{xs} \ [] = \{\} \) 
by \((\text{simp-all add: inversions-between-def})\)

We can now show the following equality for the inversions of the concatenation of two lists:

**proposition** inversions-append:

**fixes** \(\text{xs} \ \text{ys}\)
**defines** \(m \equiv \text{length \(\text{xs}\)}\) and \(n \equiv \text{length \(\text{ys}\)}\)
**shows** inversions \((\text{xs} \ @ \ \text{ys})\) =

- \(\text{inversions \(\text{xs}\)} \cup \ \text{map-prod} ((+ \ m)) \ \text{’ inversions \(\text{ys}\)} \cup \ \text{map-prod id} ((+ \ m)) \ \text{’ inversions-between \(\text{xs} \ \text{ys}\)}\)

(is - = ?rhs)

**proof** –

**note** def\(s =\) inversions-altdef inversions-between-def m-def n-def map-prod-def

**have** \(z \in \text{inversions} (\text{xs} \ @ \ \text{ys}) \longleftrightarrow z \in \ ?rhs \ \text{for} \ z\)

**proof**

assume \(z \in \text{inversions} (\text{xs} \ @ \ \text{ys})\)
then obtain \(i \ j\) where \(\text{simp:} \ z = (i, \ j)\)
and \(ij\) if \(i < j \ j < m + n\) less \(((\text{xs} @ \ \text{ys}) ! j) ((\text{xs} @ \ \text{ys}) ! i)\)
by \((\text{cases} \ z)\) \((\text{auto simp: inversions-altdef m-def n-def})\)
from \(ij\) consider \(j < m \ | \ i \geq m \ | \ i < m \ j \geq m\) by linarith
thus \(z \in \ ?rhs\)
**proof cases**

assume \(i < m \ j \geq m\)
**define** \(j' \) where \(j' = j - m\)
**have** \(\text{simp:} \ j = m + j'\)
using \(j \geq m\) by \((\text{simp add: j'-def})\)
from \(ij\) and \(i < m\) **show** \(?thesis\)
by \((\text{auto simp: inversions-altdef map-prod-def inversions-between-def nth-append m-def n-def})\)

next

assume \(i \geq m\)
**define** \(i' \ j'\) where \(i' = i - m\) and \(j' = j - m\)
**have** \(\text{simp:} \ i = m + i' \ j = m + j'\)
using \(i < j\) and \(i \geq m\) by \((\text{simp-all add: i'-def j'-def})\)
from \(ij\) **show** \(?thesis\)
by \((\text{auto simp: inversions-altdef map-prod-def nth-append m-def n-def})\)

**qed** (use \(ij\) in \((\text{auto simp: nth-append defs})\))
**qed** (\text{auto simp: nth-append defs})
**thus** \(?thesis\) by blast
**qed**

**1.2 Counting inversions**

We now define versions of \text{inversions} and \text{inversions-between} that only return the number of inversions.

**definition** inversion-number :: ‘a list ⇒ nat where
inversion-number \(\text{xs}\) = \(\text{card (inversions \(\text{xs}\))}\)
definition inversion-number-between where
  inversion-number-between xs ys = card (inversions-between xs ys)

lemma inversions-between-code:
  inversions-between xs ys =  
    Set.filter (λ(i,j). less (ys ! j) (xs ! i)) ({..<length xs}×{..<length ys})
by (auto simp: inversions-between-def)

lemmas (in −) [code] = inversions-between-code

lemma inversion-number-nil [simp]: inversion-number [] = 0
  by (simp add: inversion-number-def)

lemma inversion-number-trivial [simp]: length xs ≤ Suc 0 → inversion-number xs = 0
  by (auto simp: inversion-number-def)

lemma inversion-number-between-nil [simp]:
  inversion-number-between [] ys = 0
  inversion-number-between xs [] = 0
  by (simp-all add: inversion-number-between-def)

We again get the following nice equation for the number of inversions of a concatenation:

proposition inversion-number-append:
  inversion-number (xs @ ys) =  
  inversion-number xs + inversion-number ys + inversion-number-between xs ys
proof −
  define m n where m = length xs and n = length ys
  let ?A = inversions xs
  let ?B = map-prod ((+) m) ((+) m) ' inversions ys
  let ?C = map-prod id ((+) m) ' inversions-between xs ys

  have inversion-number (xs @ ys) = card (?A ∪ ?B ∪ ?C)
    by (simp add: inversion-number-def inversions-append m-def)
  also have ... = card (?A ∪ ?B) + card ?C
    by (intro card-Un-disjoint finite-inversions finite-UnI finite-imageI)
  also have card (?A ∪ ?B) = inversion-number xs + card ?B unfolding inversion-number-def
    by (intro card-Un-disjoint finite-inversions finite-UnI finite-imageI)
  also have card ?B = inversion-number ys unfolding inversion-number-def
    by (intro card-image) (auto simp: map-prod-def inj-on-def)
  also have card ?C = inversion-number-between xs ys
    unfolding inversion-number-between-def by (intro card-image inj-onI) (auto simp: map-prod-def)
  finally show ?thesis .

4
1.3 Stability of inversions between lists under permutations

A crucial fact for counting list inversions with merge sort is that the number of inversions between two lists does not change when the lists are permuted. This is true because the set of inversions commutes with the act of permuting the list:

**lemma inversions-between-permute1**: 
assumes \( \pi \) permutes \( \{..<\text{length } xs\} \)
shows inversions-between \((\text{permute-list } \pi \text{ } xs)\) \(ys\) =
map-prod \((\text{inv } \pi) \text{ } \text{id} \) \(\text{'} \) inversions-between \(xs\) \(ys\)

**proof** –
from assms have \([simp]: \pi \ i < \text{length } xs \text{ if } i < \text{length } xs \pi \text{ permutes \(\{..<\text{length } xs\}\)}\)
for \(i \pi\)
using permutes-in-image[OF that(2)] that by auto
have \(\ast: \text{inv } \pi \text{ permutes \(\{..<\text{length } xs\}\)}\)
using assms by (rule permutes-inv)
from assms \(\ast\) show \(\text{thesis unfolding inversions-between-def map-prod-def}\)
by (force simp: image-iff permute-list-nth permutes-inverses intro: exI[of - \pi \ i for \(i\)])
qed

**lemma inversions-between-permute2**: 
assumes \( \pi \) permutes \( \{..<\text{length } ys\} \)
shows inversions-between \(xs\) \((\text{permute-list } \pi \text{ } ys)\) =
map-prod \(\text{id} \text{ } (\text{inv } \pi) \text{ } \text{'} \) inversions-between \(xs\) \(ys\)

**proof** –
from assms have \([simp]: \pi \ i < \text{length } ys \text{ if } i < \text{length } ys \pi \text{ permutes \(\{..<\text{length } ys\}\)}\)
for \(i \pi\)
using permutes-in-image[OF that(2)] that by auto
have \(\ast: \text{inv } \pi \text{ permutes \(\{..<\text{length } ys\}\)}\)
using assms by (rule permutes-inv)
from assms \(\ast\) show \(\text{thesis unfolding inversions-between-def map-prod-def}\)
by (force simp: image-iff permute-list-nth permutes-inverses intro: exI[of - \pi \ i for \(i\)])
qed

**proposition inversions-between-permute**: 
assumes \( \pi 1 \) permutes \( \{..<\text{length } xs\} \) and \( \pi 2 \) permutes \( \{..<\text{length } ys\} \)
shows inversions-between \((\text{permute-list } \pi 1 \text{ } xs)\) \((\text{permute-list } \pi 2 \text{ } ys)\) =
map-prod \((\text{inv } \pi 1) \text{ } (\text{inv } \pi 2) \text{ } \text{'} \) inversions-between \(xs\) \(ys\)
by (simp add: inversions-between-permute1 inversions-between-permute2 assms map-prod-def image-image case-prod-unfold)

**corollary inversion-number-between-permute**: 
assumes \( \pi 1 \) permutes \( \{..<\text{length } xs\} \) and \( \pi 2 \) permutes \( \{..<\text{length } ys\} \)
shows inversion-number-between \((\text{permute-list } \pi 1 \text{ } xs)\) \((\text{permute-list } \pi 2 \text{ } ys)\) =
proof
  have inversion-number-between (permute-list π1 xs) (permute-list π2 ys) =
    card (map-prod (inv π1) (inv β) inversions-between xs ys)
    by (simp add: inversion-number-between-def inversions-between-permute assms)
  also have ... = inversion-number-between xs ys
  unfolding inversion-number-between-def using assms[THEN permutes-inj-on[OF permutes-inv]]
    by (intro card-image inj-onI) (auto simp: map-prod-def)
finally show ?thesis.
qed

The following form of the above theorem is nicer to apply since it has the
form of a congruence rule.
corollary inversion-number-between-cong-mset:
  assumes mset xs = mset xs' and mset ys = mset ys'
  shows inversion-number-between xs ys = inversion-number-between xs' ys'
proof
  obtain π1 π2 where π12: π1 permutes {..<length xs'} xs = permute-list π1 xs'
    π2 permutes {..<length ys'} ys = permute-list π2 ys'
    using assms[THEN mset-eq-permutation] by metis
  thus ?thesis by (simp add: inversion-number-between-permute)
qed

1.4 Inversions between sorted lists

Another fact that is crucial to the efficient computation of the inversion
number is this: If we have two sorted lists, we can reduce computing the
inversions by inspecting the first elements and deleting one of them.
lemma inversions-between-Cons-Cons:
  assumes sorted-wrt less-eq (x # xs) and sorted-wrt less-eq (y # ys)
  shows inversion-between (x # xs) (y # ys) =
    (if ¬less y x then
      map-prod Suc id inversions-between xs (y # ys)
    else
      {..<length (x#xs)} × {0} ∪
      map-prod id Suc inversions-between (x # xs) ys)
  using assms unfolding inversions-between-def map-prod-def
  by (auto simp: set-cone-nth nth-Cons less-le-not-le image-iff
      intro: order-trans split: nat.splits)

This leads to the following analogous equation for counting the inversions
between two sorted lists. Note that a single step of this only takes constant
time (assuming we pre-computed the lengths of the lists) so that the entire
function runs in linear time.
lemma inversion-number-between-Cons-Cons:
  assumes sorted-wrt less-eq (x # xs) and sorted-wrt less-eq (y # ys)
shows \( \text{inversion-number-between} (x \neq xs) (y \neq ys) = \\ (\text{if} \neg\text{less } y \ x \ \text{then} \\ \text{inversion-number-between } xs (y \neq ys) \\ \text{else} \\ \text{inversion-number-between} (x \neq xs) \ ys + \text{length} (x \neq xs)) \)

proof (cases less \(y\ x\))

case False

hence \( \text{inversion-number-between} (x \neq xs) (y \neq ys) = \\ \text{card} (\text{map-prod} \ \text{Suc} \ \text{id} \ \text{'} \ \text{inversions-between} \ xs (y \neq ys)) \)

by (simp add: inversion-number-between-def inversions-between-Cons-Cons[OF assms])

also have \(\ldots = \text{inversion-number-between } xs (y \neq ys)\)

unfolding inversion-number-between-def by (intro card-image inj-onI) (auto simp: map-prod-def)

finally show \(\text{thesis using} \ False \ \text{by simp}\)

next

case True

hence \( \text{inversion-number-between} (x \neq xs) (y \neq ys) = \\ \text{card} (\{..<\text{length} (x \neq xs)\} \times \{0\} \cup \text{map-prod} \ \text{id} \ \text{Suc} \ \text{'} \ \text{inversions-between} (x \neq xs) \ ys) \)

by (simp add: inversion-number-between-def inversions-between-Cons-Cons[OF assms])

also have \(\ldots = \text{length} (x \neq xs) + \text{card} (\text{map-prod} \ \text{id} \ \text{Suc} \ \text{'} \ \text{inversions-between} (x \neq xs) \ ys)\)

by (subst card-Un-disjoint) auto

also have \(\text{card} (\text{map-prod} \ \text{id} \ \text{Suc} \ \text{'} \ \text{inversions-between} (x \neq xs) \ ys) = \\ \text{inversion-number-between} (x \neq xs) \ ys\)

unfolding inversion-number-between-def by (intro card-image inj-onI) (auto simp: map-prod-def)

finally show \(\text{thesis using} \ True \ \text{by simp}\)

qed

We now define a function to compute the inversion number between two lists that are assumed to be sorted using the equalities we just derived.

fun inversion-number-between-sorted :: "'a list ⇒ 'a list ⇒ nat"

where

\( \text{inversion-number-between-sorted} [] \ ys = 0 \) 

| \( \text{inversion-number-between-sorted} xs [] = 0 \) 

| \( \text{inversion-number-between-sorted} (x \neq xs) (y \neq ys) = \\ (\text{if} \neg\text{less } y \ x \ \text{then} \\ \text{inversion-number-between-sorted} xs (y \neq ys) \\ \text{else} \\ \text{inversion-number-between-sorted} (x \neq xs) \ ys + \text{length} (x \neq xs)) \)

theorem inversion-number-between-sorted-correct:

sorted-wrt less-eq xs ⇒ sorted-wrt less-eq ys ⇒ 

\( \text{inversion-number-between-sorted} xs \ ys = \text{inversion-number-between} xs \ ys \) 

by (induction xs ys rule: inversion-number-between-sorted.induct) 

(simp-all add: inversion-number-between-Cons-Cons)
1.5 Merge sort

For convenience, we first define a simple merge sort that does not compute the inversions. At this point, we need to start assuming a linear ordering since the merging function does not work otherwise.

```
context linorder
begin

definition split-list
  where split-list xs = (let n = length xs div 2 in (take n xs, drop n xs))

fun merge-lists :: 'a list ⇒ 'a list ⇒ 'a list where
  merge-lists [] ys = ys
| merge-lists xs [] = xs
| merge-lists (x # xs) (y # ys) =
  (if less-eq x y then x # merge-lists xs (y # ys) else y # merge-lists (x # xs) ys)

lemma set-merge-lists [simp]: set (merge-lists xs ys) = set xs ∪ set ys
  by (induction xs ys rule: merge-lists.induct) auto

lemma mset-merge-lists [simp]: mset (merge-lists xs ys) = mset xs + mset ys
  by (induction xs ys rule: merge-lists.induct) auto

lemma sorted-merge-lists [simp, intro]:
  sorted xs ⇒ sorted ys ⇒ sorted (merge-lists xs ys)
  by (induction xs ys rule: merge-lists.induct) auto

fun merge-sort :: 'a list ⇒ 'a list where
  merge-sort xs =
    (if length xs ≤ 1 then xs else
      merge-lists (merge-sort (take (length xs div 2) xs)) (merge-sort (drop (length xs div 2) xs)))

lemmas [simp del] = merge-sort.simps

lemma merge-sort-trivial [simp]: length xs ≤ Suc 0 ⇒ merge-sort xs = xs
  by (subst merge-sort.simps) auto

theorem mset-merge-sort [simp]: mset (merge-sort xs) = mset xs
  by (induction xs rule: merge-sort.induct)
    (subst merge-sort.simps, auto simp flip: mset-append)

corollary set-merge-sort [simp]: set (merge-sort xs) = set xs
```
by (rule mset-eq-setD) simp-all

theorem sorted-merge-sort [simp, intro]: sorted (merge-sort xs)
  by (induction xs rule: merge-sort.induct)
    (subst merge-sort.simps, use sorted01 in auto)

lemma inversion-number-between-code:
  inversion-number-between xs ys = inversion-number-between-sorted (sort xs) (sort ys)
  by (subst inversion-number-between-sorted-correct)
    (simp-all add: sorted-sorted-wrt [symmetric] cong: inversion-number-between-cong-mset)

lemmas (in −) [code-unfold] = inversion-number-between-code

1.6 Merge sort with inversion counting

Finally, we can put together all the components and define a variant of merge sort that counts the number of inversions in the original list:

function sort-and-count-inversions :: 'a list ⇒ 'a list × nat where
  sort-and-count-inversions xs =
    (if length xs ≤ 1 then
      (xs, 0)
    else
      let (xs1, xs2) = split-list xs;
        (xs1', m) = sort-and-count-inversions xs1;
        (xs2', n) = sort-and-count-inversions xs2
      in
        (merge-lists xs1' xs2', m + n + inversion-number-between-sorted xs1' xs2'))
  by auto
termination by (relation measure length) (auto simp: split-list-def Let-def)

lemmas [simp del] = sort-and-count-inversions.simps

The projection of this function to the first component is simply the standard merge sort algorithm that we defined and proved correct before.

theorem fst-sort-and-count-inversions [simp]:
  fst (sort-and-count-inversions xs) = merge-sort xs
  by (induction xs rule: length-induct)
    (subst sort-and-count-inversions.simps, subst merge-sort.simps, simp-all add: split-list-def case-prod-unfold Let-def)

The projection to the second component is the inversion number.

theorem snd-sort-and-count-inversions [simp]:
  snd (sort-and-count-inversions xs) = inversion-number xs
proof (induction xs rule: length-induct)
  case (1 xs)
  show ?case
proof (cases length xs ≤ 1)
  case False
  have xs = take (length xs div 2) xs @ drop (length xs div 2) xs by simp
  also have inversion-number ... = snd (sort-and-count-inversions xs)
    by (subst inversion-number-append, subst sort-and-count-inversions.simps)
    (use False in (auto simp: Let-def split-list-def case-prod-unfold
      inversion-number-between-sorted-correct
      sorted-sorted-wrt [symmetric]
      cong: inversion-number-between-cong-mset))
  finally show ?thesis ..
  qed (auto simp: sort-and-count-inversions.simps)
qed

lemmas (in −) [code-unfold] = snd-sort-and-count-inversions [symmetric]

end

end

References