The Inversions of a List

Manuel Eberl

March 17, 2025

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

theory *List-Inversions*
imports
  Main
  HOL−Combinatorics.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*} (λ*i*. {*i*<..<*length xs*})

⟨*proof*⟩

lemma *finite-inversions* [*intro*]: *finite* (*inversions xs*)
⟨*proof*⟩

lemma *inversions-altdef*: *inversions xs* = {(*i*, *j*). *i* < *j* ∧ *j* < *length xs* ∧ *less* (*xs* ! *j*) (*xs* ! *i*)}
⟨*proof*⟩

lemma *inversions-code*:
inversions xs =
Sigma {..<*length xs*} (λ*i*. *Set.filter* (λ*j*. *less* (*xs* ! *j*) (*xs* ! *i*)) {*i*<..<*length xs*})
⟨*proof*⟩

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

lemma *inversions-trivial* [*simp*]: *length xs* ≤ *Suc 0* ⟹ *inversions xs* = {}
⟨*proof*⟩

lemma *inversions-imp-less*:
z ∈ *inversions xs* ⟹ *fst z* < *snd z*
z ∈ *inversions xs* ⟹ *snd z* < *length xs*
⟨*proof*⟩

lemma *inversions-Nil* [*simp*]: *inversions* [] = {}
⟨*proof*⟩

lemma *inversions-Cons*:
inversions (*x* # *xs*) =
(λ*j*. (*0*, *j* + *1*)) ‘ {*j*∈{..<*length xs*}. *less* (*xs* ! *j*) *x*} ∪
map-prod Suc Suc ‘ *inversions xs* (**is** - = *?rhs*)
⟨*proof*⟩

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*) ∈ {..<*length xs*}×{..<*length ys*}. *less* (*ys* ! *j*) (*xs* ! *i*)}

lemma *finite-inversions-between* [*intro*]: *finite* (*inversions-between xs ys*)
⟨*proof*⟩

lemma *inversions-between-Nil* [*simp*]:
inversions-between [] *ys* = {}
inversions-between xs [] = {}
⟨*proof*⟩

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

proposition *inversions-append*:
fixes *xs ys*
defines $m \equiv$ *length xs* **and** $n \equiv$ *length ys*
shows *inversions* (*xs* @ *ys*) =
inversions xs ∪ *map-prod* ((+) *m*) ((+) *m*) ' *inversions ys* ∪
map-prod id ((+) *m*) ' *inversions-between xs ys*
(**is** - = *?rhs*)
⟨*proof*⟩

1.2 Counting inversions

We now define versions of *inversions* and *inversions-between* that only return the *number* of inversions.

definition *inversion-number* :: *'a list* ⇒ *nat* **where**
inversion-number xs = *card* (*inversions 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*})
⟨*proof*⟩

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

lemma *inversion-number-Nil* [*simp*]: *inversion-number* [] = *0*
⟨*proof*⟩

lemma *inversion-number-trivial* [*simp*]: *length xs* ≤ *Suc 0* ⟹ *inversion-number xs* = *0*
⟨*proof*⟩

lemma *inversion-number-between-Nil* [*simp*]:
inversion-number-between [] *ys* = *0*
inversion-number-between xs [] = *0*
⟨*proof*⟩

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*⟩

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 π *permutes* $\{..<$*length xs*$\}$
  shows *inversions-between* (*permute-list* π *xs*) *ys* =
    map-prod (*inv* π) *id* ‘ *inversions-between xs ys*
⟨*proof*⟩

lemma *inversions-between-permute2*:
  assumes π *permutes* $\{..<$*length ys*$\}$
  shows *inversions-between xs* (*permute-list* π *ys*) =
    map-prod id (*inv* π) ‘ *inversions-between xs ys*
⟨*proof*⟩

proposition *inversions-between-permute*:
  assumes $\pi 1$ *permutes* $\{..<$*length xs*$\}$ **and** $\pi 2$ *permutes* $\{..<$*length ys*$\}$
  shows *inversions-between* (*permute-list* $\pi 1$ *xs*) (*permute-list* $\pi 2$ *ys*) =
    map-prod (*inv* $\pi 1$) (*inv* $\pi 2$) ‘ *inversions-between xs ys*
  ⟨*proof*⟩

corollary *inversion-number-between-permute*:
  assumes $\pi 1$ *permutes* $\{..<$*length xs*$\}$ **and** $\pi 2$ *permutes* $\{..<$*length ys*$\}$
  shows *inversion-number-between* (*permute-list* $\pi 1$ *xs*) (*permute-list* $\pi 2$ *ys*) =
    inversion-number-between xs ys
⟨*proof*⟩

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*⟩

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 *inversions-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)
  ⟨proof⟩
```

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   inversion-number-between (x # xs) (y # ys) =
           (if ¬less y x then
              inversion-number-between xs (y # ys)
            else
              inversion-number-between (x # xs) ys + length (x # xs))
⟨proof⟩
```

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
  inversion-number-between-sorted [] ys = 0
| inversion-number-between-sorted xs [] = 0
| inversion-number-between-sorted (x # xs) (y # ys) =
           (if ¬less y x then
              inversion-number-between-sorted xs (y # ys)
            else
              inversion-number-between-sorted (x # xs) ys + length (x # xs))

theorem inversion-number-between-sorted-correct:
  sorted-wrt less-eq xs ⟹ sorted-wrt less-eq ys ⟹
     inversion-number-between-sorted xs ys = inversion-number-between xs ys
  ⟨proof⟩

end
```

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
  ⟨proof⟩

lemma mset-merge-lists [simp]: mset (merge-lists xs ys) = mset xs + mset ys
  ⟨proof⟩

lemma sorted-merge-lists [simp, intro]:
  sorted xs ⟹ sorted ys ⟹ sorted (merge-lists xs ys)
  ⟨proof⟩


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
  ⟨proof⟩

theorem mset-merge-sort [simp]: mset (merge-sort xs) = mset xs
  ⟨proof⟩

corollary set-merge-sort [simp]: set (merge-sort xs) = set xs
  ⟨proof⟩

theorem sorted-merge-sort [simp, intro]: sorted (merge-sort xs)
  ⟨proof⟩

lemma inversion-number-between-code:
 inversion-number-between xs ys = inversion-number-between-sorted (sort xs) (sort
ys)
  ⟨proof⟩

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'))
  ⟨proof⟩
termination ⟨proof⟩

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
  ⟨proof⟩
```

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⟩

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

end

end
```

References

[1] T. H. Cormen, C. Lee, and E. Lin. *Instructor's Manual to accompany Introduction to Algorithms, 2nd Edition.* MIT Press, 2002.