Comparison-based Sorting Algorithms

Manuel Eberl

September 13, 2023

Abstract

This article contains a formal proof of the well-known fact that the number of comparisons that a comparison-based sorting algorithm needs to perform to sort a list of length $n$ is at least $\log_2(n!)$ in the worst case, i.e. $\Omega(n \log n)$.

For this purpose, a shallow embedding for comparison-based sorting algorithms is defined: a sorting algorithm is a recursive datatype containing either a HOL function or a query of a comparison oracle with a continuation containing the remaining computation. This makes it possible to force the algorithm to use only comparisons and to track the number of comparisons made.

Contents

1 Linear orderings as relations
   1.1 Auxiliary facts ........................................ 2
   1.2 Sortedness w.r.t. a relation ............................ 2
   1.3 Linear orderings ........................................ 4
   1.4 Converting a list into a linear ordering ............. 5
   1.5 Insertion sort .......................................... 5
   1.6 Obtaining a sorted list of a given set ............... 7
   1.7 Rank of an element in an ordering ................... 9
   1.8 The bijection between linear orderings and lists ...... 12

2 Lower bound on costs of comparison-based sorting 13
   2.1 Abstract description of sorting algorithms ........... 13
   2.2 Lower bounds on number of comparisons ............... 16
1 Linear orderings as relations

theory Linorder-Relations
imports Complex-Main
    HOL-Combinatorics.Multiset-Permutations
    List-Index.List-Index
begin

1.1 Auxiliary facts

lemma distinct-count-atmost-1':
distinct xs = (∀ a. count (mset xs) a ≤ 1)
proof -
  { fix x have count (mset xs) x = (if x ∈ set xs then 1 else 0) "\(-" count (mset xs) x ≤ 1
    using count-eq-zero-iff[of mset xs x]
    by (cases count (mset xs) x) (auto simp del: count-mset-0-iff)
  }
  thus ?thesis unfolding distinct-count-atmost-1' by blast
qed

lemma distinct-mset-mono:
  assumes distinct ys mset xs ⊆# mset ys
  shows distinct xs
unfolding distinct-count-atmost-1'
proof
  fix x
  from assms(2) have count (mset xs) x ≤ count (mset ys) x
    by (rule mset-subset-eq-count)
  also from assms(1) have ... ≤ 1 unfolding distinct-count-atmost-1'..
  finally show count (mset xs) x ≤ 1 .
qed

lemma mset-eq-imp-distinct-iff:
  assumes mset xs = mset ys
  shows distinct xs ←→ distinct ys
using assms by (simp add: distinct-count-atmost-1')

lemma total-on-subset: total-on B R ⊆# A ⊆ B "\(\Rightarrow\) total-on A R
by (auto simp: total-on-def)

1.2 Sortedness w.r.t. a relation

inductive sorted-wrt :: ('a × 'a) set ⇒ 'a list ⇒ bool for R where
  sorted-wrt R []
| sorted-wrt R xs ⇒ (∀ y. y ∈ set xs ⇒ (x,y) ∈ R) "\(\Rightarrow\) sorted-wrt R (x # xs)

lemma sorted-wrt-Nil [simp]: sorted-wrt R []
by (rule sorted-wrt.intros)

lemma sorted-wrt-Cons: sorted-wrt \( R (x \# xs) \iff (\forall y \in \text{set} \ xs. (x,y) \in R) \land \) sorted-wrt \( R \) \( xs \)
by (auto intro: sorted-wrt.intros elim: sorted-wrt.cases)

lemma sorted-wrt-singleton [simp]: sorted-wrt \( R \) \( [x] \)
by (intro sorted-wrt.intros) simp-all

lemma sorted-wrt-many:
  assumes trans \( R \)
  shows sorted-wrt \( R (x \# y \# xs) \iff (x, y) \in R \land \) sorted-wrt \( R \) \( y \# xs \)
by (force intro: sorted-wrt.intros transD \[ OF assms \] elim: sorted-wrt.cases)

lemma sorted-wrt-imp-le-last:
  assumes sorted-wrt \( R \) \( xs \) \( xs \neq [] \) \( x \in \text{set} \) \( xs \) \( x \neq \text{last} \) \( xs \)
  shows \( (x, \text{last} \) \( xs \) \( ) \in R \)
using assms by induction auto

lemma sorted-wrt-append:
  assumes sorted-wrt \( R \) \( xs \) sorted-wrt \( R \) \( ys \)
\[ \forall x, y \in \text{set} \) \( xs \) \( y \in \text{set} \) \( ys \) \( (x, y) \in R \) \( \land \) \( \) trans \( R \)
  shows sorted-wrt \( R \) \( (xs @ ys) \)
using assms by (induction \( xs \)) (auto simp: sorted-wrt-Cons)

lemma sorted-wrt-snoc:
  assumes sorted-wrt \( R \) \( xs \) \( (\text{last} \) \( xs \), \( y \) \( ) \in R \) \( \land \) trans \( R \)
  shows sorted-wrt \( R \) \( (xs @ [y]) \)
using assms(1,2) proof induction
  case \( \) \( 2 \) \( xs \) \( x \)
  show ?case
    proof (cases \( xs = [] \))
      case False
      with \( 2 \) \( \) have \( (z,y) \in R \) \( \text{if} \) \( z \in \text{set} \) \( xs \) \( \text{for} \) \( z \)
      using that by (cases \( z = \text{last} \) \( xs \) \( ) \( \land \) \( \) trans \( R \)
      (auto intro: assms transD[OF assms(3), transD[OF assms(3)] elim: sorted-wrt.cases])
    \( 2(1))\)
    from False have \( *: \) \( \text{last} \) \( xs \) \( \in \text{set} \) \( xs \) \( \text{by} \) simp
    moreover from \( 2 \) \( False \) \( \) have \( (x,y) \in R \) \( \text{by} \) simp
    \( 2(2))\)
    ultimately show \( \) ?thesis using \( 2 \) \( False \)
    by (auto intro: assms simp)
qed (insert \( 2 \), auto intro: sorted-wrt.intros)

lemma sorted-wrt-conv-nth:
  sorted-wrt \( R \) \( xs \) \( \iff \) \( \forall i, j. i < j \land \) \( j < \) \( \text{length} \) \( xs \) \( \rightarrow (xs \! i, \) \( xs \! j) \in R \)\)
by (induction \( xs \)) (auto simp: sorted-wrt-Cons nth-Cons set-conv-nth split: nat.splits)
1.3 Linear orderings

definition linorder-on :: 'a set ⇒ ('a × 'a) set ⇒ bool where
linorder-on A R ⇔ refl-on A R ∧ antisym R ∧ trans R ∧ total-on A R

lemma linorder-on-cases:
  assumes linorder-on A R x ∈ A y ∈ A
  shows x = y ∨ ((x, y) ∈ R ∧ (y, x) /∈ R) ∨ ((y, x) ∈ R ∧ (x, y) /∈ R)
  using assms by (auto simp: linorder-on-def refl-on-def total-on-def antisym-def)

lemma sorted-wrt-linorder-imp-index-le:
  assumes linorder-on A R set xs ⊆ A sorted-wrt R xs
  x ∈ set xs y ∈ set xs (x,y) ∈ R
  shows index xs x ≤ index xs y
  proof (cases x = y)
    case False
    define i j where i = index xs x and j = index xs y
    from False and assms have i ≠ j by (simp add: i-def j-def)
    with (index xs x ≤ index xs y) have i < j by (simp add: i-def j-def)
    moreover from assms have j < length xs by (simp add: j-def)
    ultimately have (xs ! i, xs ! j) ∈ R using assms(3)
      by (auto simp: sorted-wrt-conv-nth)
    with assms show ?thesis by (simp-all add: i-def j-def)
  qed

lemma sorted-wrt-linorder-index-le-imp:
  assumes linorder-on A R set xs ⊆ A sorted-wrt R xs
  x ∈ set xs y ∈ set xs index xs x ≤ index xs y
  shows (x, y) ∈ R
  proof (cases x = y)
    case False
    define i j where i = index xs x and j = index xs y
    from False and assms have i ≠ j by (simp add: i-def j-def)
    with (index xs x ≤ index xs y) have i < j by (simp add: i-def j-def)
    moreover from assms have j < length xs by (simp add: j-def)
    ultimately have (xs ! i, xs ! j) ∈ R using assms(3)
      by (auto simp: sorted-wrt-conv-nth)
    with assms show ?thesis by (simp-all add: i-def j-def)
  qed (insert assms, auto simp: linorder-on-def refl-on-def)

lemma sorted-wrt-linorder-index-le-iff:
  assumes linorder-on A R set xs ⊆ A sorted-wrt R xs
  x ∈ set xs y ∈ set xs
  shows index xs x ≤ index xs y ⇔ (x, y) ∈ R
  using sorted-wrt-linorder-index-le-imp[OF assms] sorted-wrt-linorder-imp-index-le[OF assms]
  by blast

lemma sorted-wrt-linorder-index-less-iff:
assumes linorder-on $A$ $R$ set $xs \subseteq A$
shows $\text{index } xs \ x < \text{index } xs \ y \iff (y, x) \notin R$
by (subst sorted-wrt-linorder-index-le-iff[OF assms(1-3) assms(5,4), symmetric]) auto

lemma sorted-wrt-distinct-linorder-nth:
assumes linorder-on $A$ $R$ set $xs \subseteq A$ sorted-wrt $R$ $xs$ distinct $xs$
shows $i < \text{length } xs \ j < \text{length } xs$
proof (cases $i$ $j$ rule: linorder-cases)
case less
with assms show ?thesis by (simp add: sorted-wrt-conv-nth)
next
case equal
from assms have $xs!i \in \text{set } xs \ xs!j \in \text{set } xs$
by (auto simp: set-conv-nth)
with assms (2) have $xs!i \in A \ xs!j \in A$
by blast+
with linorder-on $A$ $R$ and equal show ?thesis by (simp add: linorder-on-def refl-on-def)
next
case greater
with assms have $(xs!j, xs!i) \in R$
by (auto simp add: sorted-wrt-conv-nth)
moreover from assms and greater have $xs!i \neq xs!j$
by (simp add: nth-eq-iff-index-eq)
ultimately show ?thesis
by (auto simp: linorder-on-def antisym-def)
qed

1.4 Converting a list into a linear ordering
definition linear-of-list :: '$a list \Rightarrow ('a \times 'a) set$
where linear-of-list $xs = \{(a, b). \ a \in \text{set } xs \land \ b \in \text{set } xs \land \text{index } xs \ a \leq \text{index } xs \ b\}$

lemma linear-linear-of-list [intro, simp]:
assumes distinct $xs$
shows linorder-on (set $xs$) (linear-of-list $xs$)
unfolding linorder-on-def using assms
by (auto simp: refl-on-def antisym-def trans-def total-on-def linear-of-list-def)

lemma sorted-wrt-linear-of-list [intro, simp]:
distinct $xs \Rightarrow$ sorted-wrt (linear-of-list $xs$) $xs$
by (auto simp: sorted-wrt-conv-nth linear-of-list-def index-nth-id)

1.5 Insertion sort
primrec insert-wrt :: ('a \times 'a) set \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a list
where insert-wrt $R$ $x$ [] = [x]
| insert-wrt $R$ $x$ $(y \# y)$ $ys$ = (if $(x, y) \in R$ then $x \# y \# y$ else $y \#$ insert-wrt $R$ $x$ $ys$)

lemma set-insert-wrt [simp]: set (insert-wrt $R$ $x$ $xs$) = insert $x$ (set $xs$)
by (induction xs) auto

lemma mset-insert-wrt [simp]: mset (insert-wrt R x xs) = add-mset x (mset xs)
by (induction xs) auto

lemma length-insert-wrt [simp]: length (insert-wrt R x xs) = Suc (length xs)
by (induction xs) simp-all

definition insort-wrt :: ('a × 'a) set ⇒ 'a list ⇒ 'a list
where
  insort-wrt R xs = foldr (insert-wrt R) xs []

lemma set-insort-wrt [simp]: set (insort-wrt R xs) = set xs
by (induction xs) (simp-all add: insort-wrt-def)

lemma mset-insort-wrt [simp]: mset (insort-wrt R xs) = mset xs
by (induction xs) (simp-all add: insort-wrt-def)

lemma length-insort-wrt [simp]: length (insort-wrt R xs) = length xs
by (induction xs) (simp-all add: insort-wrt-def)

lemma sorted-wrt-insert-wrt [intro]:
  linorder-on A R ⇒ set (x # xs) ⊆ A ⇒
  sorted-wrt R xs ⇒ sorted-wrt R (insert-wrt R x xs)
proof (induction xs)
  case (Cons y ys)
  from Cons.prems have (x,y) ∈ R ∨ (y,x) ∈ R
    by (cases x = y) (auto simp: linorder-on-def refl-on-def total-on-def)
  with Cons show ?case
    by (auto simp: sorted-wrt-Cons intro: transD simp: linorder-on-def)
qed auto

lemma sorted-wrt-insort [intro]:
  assumes linorder-on A R set xs ⊆ A
  shows sorted-wrt R (insert-wrt R x xs)
proof −
  from assms have set (insert-wrt R x xs) = set xs ∧ sorted-wrt R (insert-wrt R x xs)
    by (induction xs) (auto simp: insert-wrt-def intro: sorted-wrt-insert-wrt)
  thus ?thesis..
qed

lemma distinct-insort-wrt [simp]: distinct (insert-wrt R x xs) ⟷ distinct xs
by (simp add: distinct-count-atmost-1)

lemma sorted-wrt-linorder-unique:  
  assumes linorder-on A R mset xs = mset ys sorted-wrt R xs sorted-wrt R ys
  shows xs = ys
proof −
  from mset xs = mset ys have length xs = length ys by (rule mset-eq-length)
  from this and assms(2−) show ?thesis
proof (induction xs ys rule: list-induct2)
  case (Cons x xs y ys)
  have set (x # xs) = set-mset (mset (x # xs)) by simp
  also have mset (x # xs) = mset (y # ys) by fact
  also have set-mset ... = set (y # ys) by simp
  finally have eq: set (x # xs) = set (y # ys).

  have x = y
  proof (rule ccontr)
    assume x ≠ y
    with eq have x ∈ set ys y ∈ set xs by auto
    with Cons.prems and assms(1) and eq have (x, y) ∈ R (y, x) ∈ R
      by (auto simp: sorted-wrt-Cons)
    with assms(1) have x = y by (auto simp: linorder-on-def antisym-def)
    with (x ≠ y) show False by contradiction
  qed
  with Cons show ?case by (auto simp: sorted-wrt-Cons)
  qed auto

1.6 Obtaining a sorted list of a given set

definition sorted-wrt-list-of-set where
  sorted-wrt-list-of-set R A =
  (if finite A then (THE xs. set xs = A ∧ distinct xs ∧ sorted-wrt R xs) else [])

lemma mset-remdups: mset (remdups xs) = mset-set (set xs)
proof (induction xs)
  case (Cons x xs)
  thus ?case by (cases x ∈ set xs) (auto simp: insert-absorb)
  qed auto

lemma sorted-wrt-list-set:
  assumes linorder-on A R set xs ⊆ A
  shows sorted-wrt-list-of-set R (set xs) = insort-wrt R (remdups xs)
proof
  have sorted-wrt-list-of-set R (set xs) =
    (THE zsa. set zsa = set xs ∧ distinct zsa ∧ sorted-wrt R zsa)
  by (simp add: sorted-wrt-list-of-set-def)
  also have ... = insort-wrt R (remdups xs)
  proof (rule the-equality)
    fix zsa assume zsa: set zsa = set xs ∧ distinct zsa ∧ sorted-wrt R zsa
    from zsa have mset zsa = mset-set (set zsa) by (subst mset-set-set) simp-all
    also from zsa have set zsa = set xs by simp
    also have mset-set ... = mset (remdups xs) by (simp add: mset-remdups)
    finally show zsa = insort-wrt R (remdups xs) using zsa assms
      by (intro sorted-wrt-linorder-unique[OF assms(1)])
        (auto intro!: sorted-wrt-insort)
  qed (insert assms, auto intro!: sorted-wrt-insort)
finally show thesis.

qed

lemma linorder-sorted-wrt-exists:
  assumes linorder-on A R finite B B ⊆ A
  shows ∃xs. set xs = B ∧ distinct xs ∧ sorted-wrt R xs
proof –
  from ‹finite B› obtain xs where set xs = B distinct xs
  using finite-distinct-list by blast
  hence set (insort-wrt R xs) = B distinct (insort-wrt R xs)
  moreover have sorted-wrt R (insort-wrt R xs)
  using assms set xs = B by (intro sorted-wrt-insort[OF assms(1)]) auto
  ultimately show thesis by blast
qed

lemma linorder-sorted-wrt-list-of-set:
  assumes linorder-on A R finite B B ⊆ A
  shows set (sorted-wrt-list-of-set R B) = B distinct (sorted-wrt-list-of-set R B)
proof –
  have ∃!xs. set xs = B ∧ distinct xs ∧ sorted-wrt R xs
  proof (rule ex-ex1I)
    show ∃xs. set xs = B ∧ distinct xs ∧ sorted-wrt R xs
    by (rule linorder-sorted-wrt-exists assms)+
  next
    fix xs ys assume set xs = B ∧ distinct xs ∧ sorted-wrt R xs
      set ys = B ∧ distinct ys ∧ sorted-wrt R ys
    thus xs = ys
    by (intro sorted-wrt-linorder-unique[OF assms(1)]) (auto simp: set-eq-iff-mset-eq-distinct)
  qed
  from theI[OF this] show set (sorted-wrt-list-of-set R B) = B
distinct (sorted-wrt-list-of-set R B) sorted-wrt R (sorted-wrt-list-of-set R B)
  by (simp-all add: sorted-wrt-list-of-set-def ‹finite B›)
qed

lemma sorted-wrt-list-of-set-eqI:
  assumes linorder-on B R A ⊆ B set xs = A distinct xs sorted-wrt R xs
  shows sorted-wrt-list-of-set R A = xs
proof (rule sorted-wrt-linorder-unique)
  show linorder-on B R by fact
  let ?ys = sorted-wrt-list-of-set R A
  have fin [simp]; finite A by (simp-all add: assms(3) [symmetric])
  have*: distinct ?ys set ?ys = A sorted-wrt R ?ys
  by (rule linorder-sorted-wrt-list-of-set[OF assms(1)] fin assms)+
  from assms * show mset ?ys = mset xs
  by (subst set-eq-iff-mset-eq-distinct [symmetric]) simp-all
  show sorted-wrt R ?ys by fact
qed fact+
1.7 Rank of an element in an ordering

The ‘rank’ of an element in a set w.r.t. an ordering is how many smaller elements exist. This is particularly useful in linear orders, where there exists a unique n-th element for every n.

definition linorder-rank where
linorder-rank R A x = card {y∈A−{x}. (y,x) ∈ R}

lemma linorder-rank-le:
  assumes finite A
  shows linorder-rank R A x ≤ card A
  unfolding linorder-rank-def using assms
  by (rule card-mono) auto

lemma linorder-rank-less:
  assumes finite A x ∈ A
  shows linorder-rank R A x < card A
proof −
  have linorder-rank R A x ≤ card (A − {x})
    unfolding linorder-rank-def using assms by (intro card-mono) auto
  also from assms have . . . < card A by (intro psubset-card-mono) auto
  finally show ?thesis .
qed

lemma linorder-rank-union:
  assumes finite A finite B A ∩ B = {}
  shows linorder-rank R (A ∪ B) x = linorder-rank R A x + linorder-rank R B x
proof −
  have linorder-rank R (A ∪ B) x = card {y∈(A∪B)−{x}. (y,x) ∈ R}
    by (simp add: linorder-rank-def)
  also have {y∈(A∪B)−{x}. (y,x) ∈ R} = {y∈A−{x}. (y,x) ∈ R} ∪ {y∈B−{x}. (y,x) ∈ R} by blast
  also have card . . . = linorder-rank R A x + linorder-rank R B x unfolding linorder-rank-def
  using assms by (intro card-Un-disjoint) auto
  finally show ?thesis .
qed

lemma linorder-rank-empty [simp]: linorder-rank R {} x = 0
  by (simp add: linorder-rank-def)

lemma linorder-rank-singleton:
  linorder-rank R {y} x = (if x ≠ y ∧ (y,x) ∈ R then 1 else 0)
proof −
  have linorder-rank R {y} x = card {z∈{y}−{x}. (z,x) ∈ R} by (simp add: linorder-rank-def)
  also have {z∈{y}−{x}. (z,x) ∈ R} = (if x ≠ y ∧ (y,x) ∈ R then {y} else {})
    by auto
also have \( \text{card} \ldots = (\text{if } x \neq y \land (y,x) \in R \text{ then } 1 \text{ else } 0) \) by simp

finally show \(?thesis\).

qed

lemma linorder-rank-insert:
  assumes finite A y \notin A
  shows linorder-rank R (insert y A) x = 
    (if x \neq y \land (y,x) \in R \text{ then } 1 \text{ else } 0) + linorder-rank R A x
  using linorder-rank-union[of \( \{ y \} \) A x] assms by (auto simp: linorder-rank-singleton)

lemma linorder-rank-mono:
  assumes linorder-on B R finite A A \subseteq B \( x, y \in R \)
  shows linorder-rank R A x \leq linorder-rank R A y
  unfolding linorder-rank-def
  proof (rule card-mono)
    from assms have trans: trans R by (simp add: linorder-on-def)
    from assms antisym: antisym R by (simp add: linorder-on-def refl-on-def)
    from this and \( (y,x) \in R \) have \( \{ z \in A - \{ y \}. (z, y) \in R \} \subseteq \{ z \in A - \{ x \}. (z,x) \in R \} \)
      by (auto intro: transD[OF trans simp: antisym-def)
    moreover from \* and assms have \( y \notin \{ z \in A - \{ y \}. (z, y) \in R \} \)
      by auto
    ultimately have \( \{ z \in A - \{ y \}. (z, y) \in R \} \subseteq \{ z \in A - \{ x \}. (z,x) \in R \} \)
      by blast
    thus \(?thesis\) using assms unfolding linorder-rank-def by (intro psubset-card-mono)
    auto
  qed

lemma linorder-rank-le-iff:
  assumes linorder-on B R finite A A \subseteq B \( x \in A \ y \in A \)
  shows linorder-rank R A x \leq linorder-rank R A y \iff \( (x, y) \in R \)
  proof (cases x = y)
    case True
    with assms show \(?thesis\) by (auto simp: linorder-on-def refl-on-def)
  next
    case False
    from assms(1) have trans: trans R by (simp add: linorder-on-def)
    from assms have \( x \in B \ y \in B \) by auto
with (linorder-on B R) and False have \(((x, y) \in R \land (y, x) \notin R) \lor ((y, x) \in R \land (x, y) \notin R)\)
by (fastforce simp: linorder-on-def antisym-def total-on-def)
thus thesis
proof
  assume \((x, y) \in R \land (y, x) \notin R\)
  with assms show thesis by (auto intro!: linorder-rank-mono)
next
  assume \*: \((y, x) \in R \land (x, y) \notin R\)
  with linorder-rank-strict-mono[OF assms(1-3), of y x] assms False
  show thesis by auto
qed

lemma linorder-rank-eq-iff:
assumes linorder-on B R finite A A \(\subseteq\) B \(\subseteq\) x \(\in\) A y \(\in\) A
shows linorder-rank R A x = linorder-rank R A y \(\iff\) x = y
proof
  assume linorder-rank R A x = linorder-rank R A y
  with linorder-rank-le-iff[OF assms(1-5)] linorder-rank-le-iff[OF assms(1-3) assms(5,4)]
  have \((x, y) \in R \land (y, x) \notin R\)
  with assms show \(x = y\) by (auto simp: linorder-on-def antisym-def)
qed simp-all

lemma linorder-rank-set-sorted-wrt:
assumes linorder-on B R set xs \(\subseteq\) B sorted-wrt R xs x \(\in\) set xs distinct xs
shows linorder-rank R (set xs) x = index xs x
proof
  define j where \(j = \text{index} \, \text{xs} \, x\)
  from assms have \(j < \text{length} \, \text{xs}\) by (simp add: j-def)
  have \*: \(x = y \lor ((x, y) \in R \land (y, x) \notin R) \lor ((y, x) \in R \land (x, y) \notin R)\) if \(y \in \text{set} \, \text{xs} \, \text{for} \, y\)
  using linorder-on-cases[OF assms(1), of x y] assms that by auto
  from assms have \(\{y \in \text{set} \, \text{xs} - \{x\}. \, (y, x) \in R\} = \{y \in \text{set} \, \text{xs} - \{x\}. \, \text{index} \, \text{xs} \, y < \text{index} \, \text{xs} \, x\}\)
  by (auto simp: sorted-wrt-linorder-index-less-iff[OF assms(1-3)] dest: *)
  also have \(\ldots = \{\chi i. \, \text{xs} \, i \, \xi \, (i, i < j)\}\) by (auto simp: j-def)
  also have \(\ldots = (\lambda i. \, \text{xs} \, i \, \xi \, \{i, i \neq j\}\) by simp
  proof safe
  fix y assume y \(\in\) set xs index xs y \(\neq\) j
  moreover from this and j have \(y = xs \, i \, \xi \, \text{index} \, \text{xs} \, y \neq j\) by simp
  ultimately show \(y \in \{i \mid (i, i < j)\}\) by blast
  qed
  from assms and j have card \(\ldots = \text{card} \, \{i, i < j\}\)
  by (intro card-image) (auto simp: inj-on-def nth-eq-iff-index-eq)
  also have \(\ldots = j\) by simp
  finally show thesis by (simp only: j-def linorder-rank-def)
qed
lemma bij-betw-linorder-rank:
assumes linorder-on B R finite A A ⊆ B
shows bij-betw (linorder-rank R A) A {..<card A}
proof –
  define xs where xs = sorted-wrt-list-of-set R A
note xs = linorder-sorted-wrt-list-of-set[OF assms, folded xs-def]
  from ‹distinct xs› have len-xs: length xs = card A
  by (subst ‹set xs = A› [symmetric]) (auto simp: distinct-card)
  have rank: linorder-rank R (set xs) x = index xs x if x ∈ A for x
  using linorder-rank-set-sorted-wrt[OF assms 1, of xs x] asms that xs
  by simp-all
  from xs len-xs show ?thesis
  by (intro bij-betw-byWitness[where f' = λi. xs ! i])
qed

1.8 The bijection between linear orderings and lists

theorem bij-betw-linorder-of-list:
assumes finite A
shows bij-betw linorder-of-list (permutations-of-set A) {R. linorder-on A R}
proof (intro bij-betw-byWitness[where f' = λR. sorted-wrt-list-of-set R A] ballI subsetI, goal-cases)
  case (1 xs)
  thus ?case by (intro sorted-wrt-list-of-set-eqI) (auto simp: permutations-of-set-def)
  next
  case (2 R)
  hence R: linorder-on A R by simp
  from R have in-R: x ∈ A y ∈ A if (x,y) ∈ R for x y using that
  by (auto simp: linorder-on-def refl-on-def)
  let ?xs = sorted-wrt-list-of-set R A
  have xs: distinct ?xs set ?xs = A sorted-wrt R ?xs
  by (rule linorder-sorted-wrt-list-of-set[OF R] asms order refl)+
  thus ?case using sorted-wrt-linorder-index-le-iff[OF R, of ?xs]
  by (auto simp: linorder-of-list-def dest: in-R)
  next
  case (4 xs)
  then obtain R where R: linorder-on A R and xs [simp]: xs = sorted-wrt-list-of-set R A by auto
  let ?xs = sorted-wrt-list-of-set R A
  have xs: distinct ?xs set ?xs = A sorted-wrt R ?xs
  by (rule linorder-sorted-wrt-list-of-set[OF R] asms order refl)+
  thus ?case by auto
qed (auto simp: permutations-of-set-def)

corollary card-finite-linorders:
assumes finite A
shows \[ \text{card } \{ R. \text{linorder-on } A R \} = \text{fact} \left( \text{card } A \right) \]

proof

have \[ \text{card } \{ R. \text{linorder-on } A R \} = \text{card} \left( \text{permutations-of-set } A \right) \]
  by (rule sym, rule bij-betw-same-card [OF bij-betw-linorder-of-list [OF assms]])
also from assms have \[ \ldots = \text{fact} \left( \text{card } A \right) \] by (rule card-permutations-of-set)
finally show \(?thesis\).
qed

end

2 Lower bound on costs of comparison-based sorting

theory Comparison-Sort-Lower-Bound
  imports
    Complex-Main
    Linorder-Relations
    Stirling-Formula, Stirling-Formula
    Landau-Symbols, Landau-More
begin

2.1 Abstract description of sorting algorithms

We have chosen to model a sorting algorithm in the following way: A sorting algorithm takes a list with distinct elements and a linear ordering on these elements, and it returns a list with the same elements that is sorted w. r. t. the given ordering.

The use of an explicit ordering means that the algorithm must look at the ordering, i. e. it has to use pair-wise comparison of elements, since all the information that is relevant for producing the correct sorting is in the ordering; the elements themselves are irrelevant.

Furthermore, we record the number of comparisons that the algorithm makes by not giving it the relation explicitly, but in the form of a comparison oracle that may be queried.

A sorting algorithm (or ‘sorter’) for a fixed input list (but for arbitrary orderings) can then be written as a recursive datatype that is either the result (the sorted list) or a comparison query consisting of two elements and a continuation that maps the result of the comparison to the remaining computation.

datatype ‘a sorter = Return ‘a list | Query ‘a ‘a bool ⇒ ‘a sorter

Cormen et al. [1] use a similar ‘decision tree’ model where an sorting algorithm for lists of fixed size \( n \) is modelled as a binary tree where each node is a comparison of two elements. They also demand that every leaf in the tree be reachable in order to avoid ‘dead’ subtrees (if the algorithm makes

13
redundant comparisons, there may be branches that can never be taken).

Then, the worst-case number of comparisons made is simply the height of
the tree.

We chose a subtly different model that does not have this restriction on the
algorithm but instead uses a more semantic way of counting the worst-case
number of comparisons: We simply use the maximum number of comparisons
that occurs for any of the (finitely many) inputs.

We therefore first define a function that counts the number of queries for a
specific ordering and then a function that counts the number of queries in
the worst case (ranging over a given set of allowed orderings; typically, this
will be the set of all linear orders on the list).

```plaintext
primrec count-queries :: ('a × 'a) set ⇒ 'a sorter ⇒ nat where
count-queries - (Return _) = 0 |
count-queries R (Query a b f) = Suc (count-queries R (f ((a, b) ∈ R)))
```

```plaintext
definition count-wc-queries :: ('a × 'a) set set ⇒ 'a sorter ⇒ nat where
count-wc-queries Rs sorter = (if Rs = {} then 0 else Max ((λR. count-queries R sorter) ' Rs))
```

```plaintext
lemma count-wc-queries-empty [simp]: count-wc-queries {} sorter = 0
  by (simp add: count-wc-queries-def)
```

```plaintext
lemma count-wc-queries-aux:
  assumes ∀R. R ∈ Rs ⇒ sorter = sorter' R Rs ⊆ Rs' finite Rs'
  shows count-wc-queries Rs sorter ≤ Max ((λR. count-queries R (sorter' R)) ' Rs')
proof (cases Rs = {})
case False
  hence count-wc-queries Rs sorter = Max ((λR. count-queries R sorter) ' Rs)
  by (simp add: count-wc-queries-def)
also have (λR. count-queries R sorter) ' Rs = (λR. count-queries R (sorter' R)) ' Rs
  by (intro image-cong refl) (simp-all add: assms)
also have Max . . . ≤ Max ((λR. count-queries R (sorter' R)) ' Rs') using False
  by (intro Max mono assms image mono finite image I) auto
finally show ?thesis .
qed simp-all
```

```plaintext
primrec eval-sorter :: ('a × 'a) set ⇒ 'a sorter ⇒ 'a list where
eval-sorter - (Return ys) = ys |
eval-sorter R (Query a b f) = eval-sorter R (f ((a, b) ∈ R))
```

We now get an obvious bound on the maximum number of different results
that a given sorter can produce.

```plaintext
lemma card-range-eval-sorter:
  assumes finite Rs
  shows card ((λR. eval-sorter R e) ' Rs) ≤ 2 ^ count-wc-queries Rs e
```

14
using assms

proof (induction e arbitrary: Rs)
  case (Return xs Rs)
  have *: \( \lambda R. \text{eval-sorter } R \text{ (Return } xs) \) \( \triangleright \) Rs = (if Rs = \{ \} then \{ \} else \{ xs \})
  by auto
  show ?case by (subst *) auto

next
  case (Query a b f Rs)
  have \( f \text{ True} \in \text{ range } f \) \( f \text{ False} \in \text{ range } f \) by simp-all
  note IH = this [THEN Query.IH]
  let \( ?Rs1 = \{ R \in Rs. (a, b) \in R \} \) and \( ?Rs2 = \{ R \in Rs. (a, b) \notin R \} \)
  let \( ?A = (\lambda R. \text{eval-sorter } R \text{ (Query } a \ b \ f) \triangleright ?Rs1) \)
  and \( ?B = (\lambda R. \text{eval-sorter } R \text{ (f False)} \triangleright ?Rs2) \)
  from Query.prems have fin: finite ?Rs1 finite ?Rs2 by simp-all
  proof (intro subsetI, elim imageE, goal-cases)
    case (1 xs R)
    thus ?case by (cases \((a, b) \in R\)) auto
  qed

show ?case
proof (cases Rs = \{ \})
  case False
  have \( \text{card } (\lambda R. \text{eval-sorter } R \text{ (Query } a \ b \ f) \triangleright ?Rs1) \leq \text{card } (?A 
  \cup ?B) \)
  by (intro card_mono finite-UnI finite-imageI fin *)
  also have \( \ldots \leq \text{card } ?A + \text{card } ?B \) by (rule card-Un-le)
  also have \( \ldots \leq 2 ^ \text{count-wc-queries } ?Rs1 \text{ (f True) } + 2 ^ \text{count-wc-queries } ?Rs2 \text{ (f False) } \)
  by (intro add_mono IH fin)
  also have \( \text{count-wc-queries } ?Rs1 \text{ (f True) } \leq \text{Max } ((\lambda R. \text{count-queries } R \text{ (f ((a,b) \in R)))} \triangleright ?Rs) \)
  by (intro count-uc-queries-aux Query.prems) auto
  also have \( \text{count-wc-queries } ?Rs2 \text{ (f False) } \leq \text{Max } ((\lambda R. \text{count-queries } R \text{ (f ((a,b) \in R)))} \triangleright ?Rs) \)
  by (intro count-uc-queries-aux Query.prems) auto
  also have \( 2 ^ \ldots + 2 ^ \ldots = (2 ^ \ldots \cdot 2 ^ \text{nat}) \) by simp
  also have \( \text{Suc } \text{Max } ((\lambda R. \text{count-queries } R \text{ (f ((a,b) \in R)))} \triangleright ?Rs) \) =
  \( \text{Max } (\text{Suc } ((\lambda R. \text{count-queries } R \text{ (f ((a,b) \in R)))} \triangleright ?Rs)) \) using False
  by (intro mono-Max-commute finite-imageI Query.prems) (auto simp: inc-seq-def)
  also have \( \text{Suc } ((\lambda R. \text{count-queries } R \text{ (f ((a,b) \in R)))} \triangleright ?Rs) =
  \( (\lambda R. \text{Suc } (\text{count-queries } R \text{ (f ((a,b) \in R)))) \triangleright ?Rs) \) by (simp add: image-image)
  also have \( \text{Max } \ldots = \text{count-wc-queries } Rs \text{ (Query } a \ b \ f) \) using False
  by (auto simp add: count-uc-queries-def)
  finally show ?thesis by - simp-all
qed simp-all

qed
The following predicate describes what constitutes a valid sorting result for a given ordering and a given input list. Note that when the ordering is linear, the result is actually unique.

definition is-sorting :: ('a × 'a) set ⇒ 'a list ⇒ 'a list ⇒ bool
where
is-sorting R xs ys ←→ (mset xs = mset ys) ∧ sorted-wrt R ys

2.2 Lower bounds on number of comparisons

For a list of \( n \) distinct elements, there are \( n! \) linear orderings on \( n \) elements, each of which leads to a different result after sorting the original list. Since a sorter can produce at most \( 2^k \) different results with \( k \) comparisons, we get the bound \( 2^k \geq n! \):

theorem
fixes sorter :: 'a sorter and xs :: 'a list
assumes distinct: distinct xs
assumes sorter: \( \forall R. \text{linorder-on} \ (\text{set} \ xs) \ R \implies \text{is-sorting} \ R \ xs \ \text{(eval-sorter} \ R \ sorter) \)
defines Rs ≡ \{ R. \text{linorder-on} \ (\text{set} \ xs) \ R \}
shows two-power-count-queries-ge: \( \text{fact} \ (\text{length} \ xs) \leq (2 ^ \text{count-wc-queries} \ Rs \ \text{sorter} :: \text{nat}) \)
and count-queries-ge: \( \log 2 \ (\text{fact} \ (\text{length} \ xs)) \leq \text{real} \ (\text{count-wc-queries} \ Rs \ \text{sorter}) \)
proof
− have Rs ⊆ Pow (set xs × set xs) by (auto simp: Rs-def linorder-on-def refl-on-def)
hence fin: finite Rs by (rule finite-subset) simp-all
from assms have \( \text{fact} \ (\text{length} \ xs) = \text{card} \ (\text{permutations-of-set} \ (\text{set} \ xs)) \)
by (simp add: distinct-card)
also have \( \text{permutations-of-set} \ (\text{set} \ xs) \subseteq (\lambda R. \text{eval-sorter} \ R \ sorter) \) \( \cdot \) Rs
proof (rule subsetI, goal-cases)
case (1 ys)
define R where R = \text{linorder-of-list} \ ys
define zs where zs = \text{eval-sorter} \ R \ sorter
from 1 and distinct have mset-ys: mset ys = mset xs
by (auto simp: set-eq-iff-mset-eq-distinct permutations-of-set-def)
from 1 have *: \text{linorder-on} \ (\text{set} \ xs) \ R \text{ unfolding} \ R-def using \text{linorder-linorder-of-list[of ys]}
by (simp add: permutations-of-set-def)
from sorter[of this] have mset xs = mset zs \text{ sorted-wrt} R zs
by (simp-all add: is-sorting-def zs-def)
moreover from 1 have \text{sorted-wrt} R \ ys \text{ unfolding} \ R-def
by (intro \text{sorted-wrt-linorder-of-list}) (simp-all add: permutations-of-set-def)
ultimately have \( zs = ys \)
by (intro \text{sorted-wrt-linorder-unique}[OF *]) (simp-all add: mset-ys)
moreover from * have R ∈ Rs by (simp add: Rs-def)
ultimately show \(?case unfolding \text{zs-def by blast}
qed
hence \text{card} \ (\text{permutations-of-set} \ (\text{set} \ xs)) \leq \text{card} \ ((\lambda R. \text{eval-sorter} \ R \ sorter) \ ) \ Rs)
This leads to the following well-known Big-Omega bound on the number of
comparisons that a general sorting algorithm has to make:

corollary count-queries-bigomega:
  fixes sorter :: nat ⇒ nat sorter
  assumes sorter: \( \forall n. \text{linorder-on } (..<n) R \implies \text{is-sorting } R [\emptyset..<n] (\text{eval-sorter } R (\text{sorter } n)) \)
  defines \( R_s \equiv \lambda n. \{R. \text{linorder-on } (..<n) R\} \)
  shows \( (\lambda n. \text{count-wc-queries } (R_s n) (\text{sorter } n)) \in \Omega(\lambda n. n \ast \ln n) \)
proof –
  have \( (\lambda n. n \ast \ln n) \in \Theta(\lambda n. \ln (\text{fact } n)) \)
    by (subst bigtheta-sym) (intro asymp-equiv-imp-bigtheta asymp-equiv-intros)
  also have \( (\lambda n. \ln (\text{fact } n)) \in \Theta(\lambda n. \log 2 (\text{fact } n)) \) by (simp add: log-def)
  also have \( (\lambda n. \log 2 (\text{fact } n)) \in O(\lambda n. \text{count-wc-queries } (R_s n) (\text{sorter } n)) \)
proof (intro bigoI [where \( c = 1 \]) always-eventually allI, goal-cases)
  case (1 n)
    have norm \((\log 2 (\text{fact } n)) = \log 2 (\text{fact } (\text{length } [\emptyset..<n]))\) by simp
    also from sorter[of n] have \( \ldots \leq \text{real } (\text{count-wc-queries } (R_s n) (\text{sorter } n)) \)
      using count-queries-ge[of [\emptyset..<n] sorter n] by (auto simp: Rs-def atLeast0LessThan)
    also have \( \ldots = 1 \ast \text{norm } \ldots \) by simp
    finally show \(?case\) by simp
  qed
  finally show \(?thesis\) by (simp add: bigomega-iff-bigo)
  qed
end

References