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 2
  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−Library.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 ⇒ 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) ⇒ sorted-wrt R (x # xs)

lemma sorted-wrt-Nil [simp]: sorted-wrt R []
lemma sorted-wrt-Cons: sorted-wrt R (x ≠ xs) ↔ (∀ y∈set xs. (x,y) ∈ R) ∧ 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) ↔ (x,y) ∈ R ∧ 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 ≠ [] x ∈ set xs x ≠ last xs
shows (x, last xs) ∈ R
using assms by induction auto

lemma sorted-wrt-append:
assumes sorted-wrt R xs sorted-wrt R ys
  \(\forall x, y. x \in set xs \implies y \in set ys \implies (x,y) \in R\)
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 (last xs, y) ∈ R 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) ∈ R if z ∈ set xs for z
    using that by (cases z = last xs)
      (auto intro: assms transD [OF assms(3), OF sorted-wrt-imp-le-last [OF 2(1)]])
  from False have *: last xs ∈ set xs by simp
    moreover from 2 False have (x,y) ∈ R by (intro transD [OF assms(3) OF *]) simp
  ultimately show ?thesis using 2 False by (auto intro!: sorted-wrt.intros)
  qed (insert 2, auto intro: sorted-wrt.intros)
qed simp-all

lemma sorted-wrt-conv-nth:
sorted-wrt R xs ↔ (∀ i, j. i < j ∧ j < length xs → (xs!i, xs!j) ∈ R)
by (induction xs) (auto simp: sorted-wrt-Cons nth-Cons set-conv-nth split: nat.splits)
1.3 Linear orderings

**Definition** \( \text{linorder-on} :: \ 'a \Rightarrow (\ 'a \times \ 'a) \Rightarrow \text{bool} \) where

\( \text{linorder-on} 
\mathit{A} 
\mathit{R} \iff \text{refl-on} \mathit{A} \mathit{R} \land \text{antisym} \mathit{R} \land \text{trans} \mathit{R} \land \text{total-on} \mathit{A} \mathit{R} \)

**Lemma** **linorder-on-cases:**

- \( \mathit{assumes} \ \text{linorder-on} \ \mathit{A} \ \mathit{R} \ \mathit{x} \in \mathit{A} \ \mathit{y} \in \mathit{A} \)
- \( \mathit{shows} \ \mathit{x} = \mathit{y} \lor (\ (\mathit{x}, \mathit{y}) \in \mathit{R} \land (\mathit{y}, \mathit{x}) \notin \mathit{R} ) \lor (\ (\mathit{y}, \mathit{x}) \in \mathit{R} \land (\mathit{x}, \mathit{y}) \notin \mathit{R} ) \)

**Lemma** **sorted-wrt-linorder-imp-index-le:**

- \( \mathit{assumes} \ \text{linorder-on} \ \mathit{A} \ \mathit{R} \ \mathit{set} \ \mathit{xs} \subseteq \mathit{A} \ \text{sorted-wrt} \ \mathit{R} \ \mathit{xs} \)
- \( \mathit{x} \in \mathit{set} \ \mathit{xs} \ \mathit{y} \in \mathit{set} \ \mathit{xs} \)
- \( (\mathit{x}, \mathit{y}) \in \mathit{R} \)
- \( \mathit{shows} \ \mathit{index} \ \mathit{xs} \ \mathit{x} \leq \mathit{index} \ \mathit{xs} \ \mathit{y} \)

**Lemma** **sorted-wrt-linorder-index-le-imp:**

- \( \mathit{assumes} \ \text{linorder-on} \ \mathit{A} \ \mathit{R} \ \mathit{set} \ \mathit{xs} \subseteq \mathit{A} \ \text{sorted-wrt} \ \mathit{R} \ \mathit{xs} \)
- \( \mathit{x} \in \mathit{set} \ \mathit{xs} \ \mathit{y} \in \mathit{set} \ \mathit{xs} \)
- \( (\mathit{x}, \mathit{y}) \in \mathit{R} \)
- \( \mathit{shows} \ \mathit{index} \ \mathit{xs} \ \mathit{x} \leq \mathit{index} \ \mathit{xs} \ \mathit{y} \)

**Lemma** **sorted-wrt-linorder-index-le-iff:**

- \( \mathit{assumes} \ \text{linorder-on} \ \mathit{A} \ \mathit{R} \ \mathit{set} \ \mathit{xs} \subseteq \mathit{A} \ \text{sorted-wrt} \ \mathit{R} \ \mathit{xs} \)
- \( \mathit{x} \in \mathit{set} \ \mathit{xs} \ \mathit{y} \in \mathit{set} \ \mathit{xs} \)
- \( (\mathit{x}, \mathit{y}) \in \mathit{R} \)
- \( \mathit{shows} \ \mathit{index} \ \mathit{xs} \ \mathit{x} \leq \mathit{index} \ \mathit{xs} \ \mathit{y} \iff (\mathit{x}, \mathit{y}) \in \mathit{R} \)

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

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

1.4 Converting a list into a linear ordering

definition \( \text{linorder-of-list} :: \{'a \ \text{list}\} \Rightarrow (\{'a \times \{'a\}\}) \ \text{set} \) where
\( \text{linorder-of-list } xs = \{(a, b). \ a \in \text{set } xs \ \wedge \ b \in \text{set } xs \ \wedge \ \text{index } xs \ a \leq \text{index } xs \ b\} \)

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

lemma \( \text{sorted-wrt-linorder-of-list} [\text{intro, simp}]: \)
distinct \( xs \implies \text{sorted-wrt } (\text{linorder-of-list } xs) \) \( xs \)
by \((\text{auto simp: } \text{sorted-wrt-conv-nth linorder-of-list-def index-nth-id})\)

1.5 Insertion sort

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

lemma \( \text{set-insert-wrt} [\text{simp}]: \text{set } (\text{insert-wrt } R \ x \ xs) = \text{insert } x \ (\text{set } xs) \)

5
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 xs) 
proof – 
  from assms have set (insert-wrt R xs) = set xs ∧ sorted-wrt R (insert-wrt R 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 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)

qed
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
qed

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) = insert-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 ... = insert-wrt R (remdups xs)
    by (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 = insert-wrt R (remdups xs) using zsa assms
    by (intro sorted-wrt-linorder-unique[OF assms(1)])
      (auto intro!: sorted-wrt-insert)
proof (insert assms, auto intro!: sorted-wrt-insert)
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 (insert-wrt R xs) = B distinct (insert-wrt R xs)
  by simp-all
  moreover have sorted-wrt R (insert-wrt R xs)
    using assms ⟨set xs = B⟩ by (intro sorted-wrt-insert[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)
  sorted-wrt R (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
\[
\text{linorder-rank } R \ A \ x = |\{ y \in A \setminus \{x\} . (y,x) \in R \}|
\]

lemma linorder-rank-le:
assumes finite \( A \)
shows \( \text{linorder-rank } R \ A \ x \leq |A| \)
unfolding linorder-rank-def using assms by (rule card-mono) auto

lemma linorder-rank-less:
assumes finite \( A \) \( x \in A \)
shows \( \text{linorder-rank } R \ A \ x < |A| \)
proof –
\( \text{have } \text{linorder-rank } R \ A \ x \leq |A - \{x\}| \)
unfolding linorder-rank-def using assms by (intro card-mono) auto
also from assms have \( \ldots \ < |A| \) by (intro psubset-card-mono) auto
finally show \( ?\text{thesis} \).

qed

lemma linorder-rank-union:
assumes finite \( A \) finite \( B \) \( A \cap B = \{} \)
shows \( \text{linorder-rank } R (A \cup B) \ x = \text{linorder-rank } R A \ x + \text{linorder-rank } R B \ x \)
proof –
\( \text{have } \text{linorder-rank } R (A \cup B) \ x = |\{ y \in (A \cup B) \setminus \{x\} . (y,x) \in R \}| \)
by (simp add: linorder-rank-def)
also have \( \{ y \in (A \cup B) \setminus \{x\} . (y,x) \in R \} = \{ y \in A \setminus \{x\} . (y,x) \in R \} \cup \{ y \in B \setminus \{x\} . (y,x) \in R \} \) by blast
also have \( \text{card} \ldots = \text{linorder-rank } R A \ x + \text{linorder-rank } R B \ x \) unfolding linorder-rank-def
using assms by (intro card-Un-disjoint) auto
finally show \( ?\text{thesis} \).

qed

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

lemma linorder-rank-singleton:
\( \text{linorder-rank } R \ \{y\} \ x = (\text{if } x \neq y \land (y,x) \in R \text{ then } 1 \text{ else } 0) \)
proof –
\( \text{have } \text{linorder-rank } R \ \{y\} \ x = |\{ z \in \{y\} \setminus \{x\} . (z,x) \in R \}| \) by (simp add: linorder-rank-def)
also have \( \{ z \in \{y\} \setminus \{x\} . (z,x) \in R \} = (\text{if } x \neq y \land (y,x) \in R \text{ then } \{y\} \text{ else } \{\}) \)
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 \( ?\text{thesis} \).
qed

lemma linorder-rank-insert:
assumes finite \( A \) \( y \notin A \)
shows \( \text{linorder-rank} \ R \ (\text{insert} \ y \ A) \ x = \)
\( (\text{if } x \neq y \land (y,x) \in R \text{ then } 1 \text{ else } 0) + \text{linorder-rank} \ R \ A \ x \)
using linorder-rank-union[of \( \{ y \} \) \( A \) \( x \)]\( \text{assms} \) 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 \( \text{linorder-rank} \ R \ A \ x \leq \text{linorder-rank} \ R \ A \ y \)
unfolding linorder-rank-def
proof (rule card-mono)
from \( \text{assms} \) have trans: trans \( R \) by (simp-add: linorder-on-def)
from \( \text{assms} \) antisym: antisym \( R \) by (intro antisym)
moreover from \( \text{trans} \) and \( \text{assms} \) have \( y \notin \{ z \in A - \{ x \}. (z,y) \in R \} \) \( y \in \{ z \in A - \{ x \}. (z,x) \in R \} \) by auto
ultimately have \( \{ z \in A - \{ y \}. (z,y) \in R \} \subset \{ z \in A - \{ x \}. (z,x) \in R \} \) by blast
thus \( ?\text{thesis} \) using \( \text{assms} \) unfolding linorder-rank-def by (intro psubset-card-mono) auto
qed

lemma linorder-rank-le-iff:
assumes linorder-on \( B \) \( R \) finite \( A \) \( x \in A \) \( y \in A \)
shows \( \text{linorder-rank} \ R \ A \ x \leq \text{linorder-rank} \ R \ A \ y \) \( \iff \) \( (x,y) \in R \)
proof (cases \( x = y \))
case True
with \( \text{assms} \) show \( ?\text{thesis} \) by (auto simp: linorder-on-def refl-on-def)
next
case False
from \( \text{assms} \) have trans: trans \( R \) by (simp-add: linorder-on-def)
from \( \text{assms} \) have \( x \in B \) \( y \in B \) by auto
with `(linorder-on B R) and False have ((x,y) ∈ R ∧ (y,x) ∉ R) ∨ ((y,x) ∈ R ∧ (x,y) ∉ R)
by (fastforce simp: linorder-on-def antisym-def total-on-def)
thus ?thesis
proof
  assume `(x,y) ∈ R ∧ (y,x) ∉ R
  with assms show ?thesis by (auto intro!: linorder-rank-mono)
next
  assume *: `(y,x) ∈ R ∧ (x,y) ∉ 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 ⊆ B x ∈ A y ∈ A
  shows linorder-rank R A x = linorder-rank R A y ←→ 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) ∈ R (y, x) ∈ R by simp-all
  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 ⊆ B sorted-wrt R xs x ∈ set xs distinct xs
  shows linorder-rank R (set xs) x = index xs x
proof –
  define j where `j = index xs x
  from assms have j: `j < length xs by (simp add: j-def)
  have *: `x = y ∨ ((x, y) ∈ R ∧ (y, x) ∉ R) ∨ ((y, x) ∈ R ∧ (x, y) ∉ R) if y ∈ set xs for y
    using linorder-on-cases[OF assms(1), of x y] assms that by auto
  from assms have `{y ∈ set xs-{x}. (y, x) ∈ R} = `{y ∈ set xs-{x}. index xs y < index xs x}
    by (auto simp: sorted-wrt-linorder-index-less-iff[OF assms(1-3)] dest: *)
  also have ‹… = {y ∈ set xs. index xs y < j}› by (auto simp: j-def)
  also have ‹… = (λi. xs ! i) ‹i. i < j››
  proof safe
    fix y assume y ∈ set xs index xs y < j
    moreover from this and j have y = xs ! index xs y by simp
    ultimately show y ∈ (!) xs ‹i. i < j› by blast
  qed (insert assms j, auto simp: index-nth-id)
  also from assms and j have card ‹… = card {i. i < j}› by (intro card-image) (auto simp: inj-on-def nth-eq-iff-index-eq)
  also have ‹… = 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] assms that xs
by simp_all
from xs len_xs show ?thesis
by (intro bij_betw_byWitness[where f' = λi. xs ! i]) (auto simp: permutations_of_set_def
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] assms 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] assms order refl)+
thus ?case by auto
qed (auto simp: permutations_of_set_def)
corollary card_finite_linorders:
assumes  finite A
shows \( \text{card} \ \{ R. \text{linorder-on} \ A \ R \} = \text{fact} (\text{card} \ A) \)

proof –

have \( \text{card} \ \{ R. \text{linorder-on} \ A \ R \} = \text{card} \ (\text{permutations-of-set} \ A) \)

by (rule sym, rule bij-betw-same-card [OF bij-betw-linorder-of-list[OF assms]]

also from assms have \( \ldots = \text{fact} \ (\text{card} \ A) \) by (rule card-permutations-of-set)

finally show \( \text{thesis} \).

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
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).

```haskell
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)))
```

```haskell
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))
```

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

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-imageI) auto
  finally show "thesis .
qed simp-all
```

```haskell
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.

```haskell
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 *: (λR. eval-sorter R (Return xs)) ' Rs = (if Rs = {[]} then {} else {xs})
  by auto
  show ?case by (subst *) auto
next
case (Query a b f Rs)
  have f True ∈ range f by simp
case (λR. eval-sorter R (Query a b f)) ' Rs ∈ range f by simp-all
note IH = this [THEN Query.IH]
let ?Rs1 = \{R ∈ Rs. (a, b) ∈ R\} and ?Rs2 = \{R ∈ Rs. (a, b) ∉ R\}
from Query.prems have fin: finite ?Rs1 finite ?Rs2 by simp-all
have *: (λR. eval-sorter R (Query a b f)) ' Rs ⊆ ?A ∪ ?B
proof (intro subsetI, elim imageE, goal-cases)
  case (1 xs R)
  then show ?case by (cases (a, b) ∈ R) auto
qed

show ?case
proof (cases Rs = {[]})
  case False
  have card ((λR. eval-sorter R (Query a b f)) ' Rs) ≤ card (?A ∪ ?B)
    by (intro card-mono finite-Union finite-imageI fin *)
  also have ... ≤ card ?A + card ?B by (rule card-Union)
  also have ... ≤ 2 ^ count-wc-queries ?Rs1 (f True) + 2 ^ count-wc-queries ?Rs2 (f False)
    by (intro add-mono IH fin)
  also have count-wc-queries ?Rs1 (f True) ≤ Max ((λR. count-queries R (f ((a,b)∈R))) ' Rs)
    by (intro count-wc-queries-aux Query.prems) auto
  also have count-wc-queries ?Rs2 (f False) ≤ Max ((λR. count-queries R (f ((a,b)∈R))) ' Rs)
    by (intro count-wc-queries-aux Query.prems) auto
  also have 2 ^ ... + 2 ^ ... = (2 ^ Suc ... :: nat) by simp
  also have Suc (Max ((λR. count-queries R (f ((a,b)∈R))) ' Rs)) =
    Max (Suc ' ((λR. count-queries R (f ((a,b)∈R))) ' Rs)) using False
    by (intro mono-Max-commute finite-imageI Query.prems) (auto simp: incseq-def)
  also have Suc ' ((λR. count-queries R (f ((a,b)∈R))) ' Rs) =
    (λR. Suc (count-queries R (f ((a,b)∈R)))) ' Rs by (simp add: image-image)
  also have Max ... = count-wc-queries Rs (Query a b f) using False
    by (auto simp add: count-wc-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**

\[
\text{is-sorting} :: (\forall \cdot a \times \cdot a) \rightarrow \cdot a \rightarrow \cdot a \rightarrow \cdot a \rightarrow \cdot bool
\]

\[
is-sorting \ R \ xs \ ys \longleftrightarrow (\text{mset} \ xs = \text{mset} \ ys) \land \text{sorted-wrt} \ R \ ys
\]

### 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**

\[
\begin{align*}
\text{fixes} & \quad \text{sorter} :: \cdot a \rightarrow \cdot a \\
\text{assumes} & \quad \text{distinct}: \text{distinct} \ xs \\
\text{assumes} & \quad \text{sorter}: \land \ R. \text{linorder-on} \ (\text{set} \ xs) \ R \implies \text{is-sorting} \ R \ xs \ (\text{eval-sorter} \ R \ \text{sorter}) \\
\text{defines} & \quad Rs \equiv \{ R. \text{linorder-on} \ (\text{set} \ xs) \ R \} \\
\text{shows} & \quad \text{two-power-count-queries-ge}: \text{fact} \ (\text{length} \ xs) \leq (2^\text{count-wc-queries} \ Rs \ \text{sorter} :: \cdot nat) \\
\text{and} & \quad \text{count-queries-ge}: \log 2 \ (\text{fact} \ (\text{length} \ xs)) \leq \text{real} \ (\text{count-wc-queries} \ Rs \ \text{sorter})
\end{align*}
\]

**proof**

\[
\begin{align*}
\text{have} & \quad Rs \subseteq \text{Pow} \ (\text{set} \ xs \times \text{set} \ xs) \ \text{by} \ (\text{auto simp: Rs-def linorder-on-def refl-on-def}) \\
\text{hence} & \quad \text{fin: finite} \ Rs \ \text{by} \ (\text{rule finite-subset simp-all}) \\
\text{from assms have} & \quad \text{fact} \ (\text{length} \ xs) = \text{card} \ (\text{permutations-of-set} \ (\text{set} \ xs)) \\
\text{by} & \quad (\text{simp add: distinct-card}) \\
\text{also have} & \quad \text{permutations-of-set} \ (\text{set} \ xs) \subseteq (\lambda R. \text{eval-sorter} \ R \ \text{sorter}) \ ' \ Rs \\
\text{proof} & \quad (\text{rule subsetI}, \text{goal-cases}) \\
\text{case} & \quad (1 \ ys) \\
\text{define} & \quad R \ \text{where} \ R = \text{linorder-of-list} \ ys \\
\text{define} & \quad zs \ \text{where} \ zs = \text{eval-sorter} \ R \ \text{sorter} \\
\text{from} & \quad 1 \ \text{and} \ \text{distinct have} \ mset-ys: mset \ ys = mset \ xs \\
\text{by} & \quad (\text{auto simp: set-eq-iff-mset-eq-distinct permutations-of-set-def}) \\
\text{from} & \quad 1 \ \text{have} \ *: \text{linorder-on} \ (\text{set} \ xs) \ R \ \text{unfolding} \ R \ \text{def using} \ \text{linorder-linorder-of-list[of ys]} \\
\text{by} & \quad (\text{simp add: permutations-of-set-def}) \\
\text{from} & \quad \text{sorter} \ [\text{OF this}] \ \text{have} \ \text{mset} \ xs = \text{mset} \ zs \ \text{sorted-wrt} \ R \ zs \\
\text{by} & \quad (\text{simp-all add: is-sorting-def zs-def}) \\
\text{moreover from} & \quad 1 \ \text{have} \ \text{sorted-wrt} \ R \ ys \ \text{unfolding} \ R \ \text{def} \\
\text{by} & \quad (\text{intro sorted-wrt-linorder-of-list}) \ (\text{simp-all add: permutations-of-set-def}) \\
\text{ultimately have} & \quad zs = ys \\
\text{by} & \quad (\text{intro sorted-wrt-linorder-unique[of *])} \ (\text{simp-all add: mset-ys}) \\
\text{moreover from} & \quad * \ \text{have} \ R \ \in \ Rs \ \text{by} \ (\text{simp add: Rs-def}) \\
\text{ultimately show} & \quad \text{case unfolding} \ zs-def \ \text{by} \ \text{blast} \\
\text{hence} & \quad \text{card} \ (\text{permutations-of-set} \ (\text{set} \ xs)) \leq \text{card} \ ((\lambda R. \text{eval-sorter} \ R \ \text{sorter}) \ ' \ Rs) \\
\text{by} & \quad (\text{intro card-mono finite-imageI fin})
\end{align*}
\]
also from fin have \ldots \leq 2 \cdot \text{count-wc-queries } Rs \text{ sorter} \text{ by (rule card-range-eval-sorter)}

finally show \ast: \text{fact (length } xs) \leq (2 \cdot \text{count-wc-queries } Rs \text{ sorter} :: \text{nat} ) .

have \ln (\text{fact (length } xs)) = \ln (\text{real (fact (length } xs))) \text{ by simp}
also have \ldots \leq \ln (\text{real (2 \cdot \text{count-wc-queries } Rs \text{ sorter})})
proof (subst ln-le-cancel-iff)
  show \ln (\text{fact (length } xs)) \leq \ln (\text{real (2 \cdot \text{count-wc-queries } Rs \text{ sorter})})
    by (subst of-nat-le-iff) (rule *)
qed simp-all
also have \ldots = \ln (\text{count-wc-queries } Rs \text{ sorter}) \ast 2 \text{ by (simp add: ln-realpow) }
finally have \ln (\text{count-wc-queries } Rs \text{ sorter}) \geq \ln (\text{fact (length } xs) ) / \ln 2
  by (simp add: field-simps)
also have \ln (\text{fact (length } xs)) / \ln 2 = \log 2 (\text{fact (length } xs)) \text{ by (simp add: log-def) }
finally show \ast\ast: \log 2 (\text{fact (length } xs)) \leq \text{real (count-wc-queries } Rs \text{ sorter}) .
qed

lemma \ln\text{-fact-bigo}: (\lambda n. \ln (\text{fact } n) - (\ln (2 * pi * n) / 2 + n * \ln n - n)) \in O(\lambda n. 1 / n)
  and asymp-equiv-ln-fact [asymp-equiv-intros]: (\lambda n. \ln (\text{fact } n)) \sim [at-top] (\lambda n. n * \ln n)
proof –
  include asymp-equiv-notation
define f where f = (\lambda n. \ln (2 * pi * \text{real } n) / 2 + \text{real } n * \ln (\text{real } n) - \text{real } n)
have eventually (\lambda n. \ln (\text{fact } n) - f n \in \{0..1/(12*\text{real } n)\}) \text{ at-top }
  using eventually-\text{gt-at-top}[\text{of } 1::\text{nat}]
proof eventually-elim
  case \text{(elim } n)
  with \ln\text{-fact-bounds}[\text{of } n] \text{ show } ?\text{case by (simp add: f-def) }
qed

hence eventually (\lambda n. \text{norm (ln (fact } n) - f n) \leq (1/12) * \text{norm (1 / real } n)) \text{ at-top }
  using eventually-\text{gt-at-top}[\text{of } 0::\text{nat}] \text{ by eventually-elim (simp-all add: field-simps) }
thus (\lambda n. \ln (\text{fact } n) - f n) \in O(\lambda n. 1 / \text{real } n)
  using \text{bigo}[\text{of } \lambda n. \ln (\text{fact } n) - f n 1/12 \lambda n. 1 / \text{real } n] \text{ by simp }
also have (\lambda n. 1 / \text{real } n) \in o(f) \text{ unfolding f-def by (intro smallo-realtat-transfer) }
simp
  finally have (\lambda n. f n + (\ln (\text{fact } n) - f n)) \sim f
    by (subst asymp-equiv-add-right) simp-all
  hence (\lambda n. \ln (\text{fact } n)) \sim f \text{ by simp }
  also have f \sim (\lambda n. n * \ln n + (\ln (2*pi*n)/2 - n)) \text{ by (simp add: f-def algebra-simps) }
also have \ldots \sim (\lambda n. n * \ln n) \text{ by (subst asymp-equiv-add-right) auto }
finally show (\lambda n. \ln (\text{fact } n)) \sim (\lambda n. n * \ln n) .
qed

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: \( \forall n. \text{linorder-on } \{..<n\} R \implies \) is-sorting \( R \{0..<n\} \) (eval-sorter \( R \) (sorter \( n \)))
- **defines** \( Rs \equiv \lambda n. \{ R. \text{linorder-on } \{..<n\} \} R \)
- **shows** \( (\lambda n. \text{count-wc-queries} \{ Rs n \} \text{ (sorter } n \}) \in \Omega(\lambda n. n \ast \ln n) \)

**proof** −

- have \( (\lambda n. n \ast \ln n) \in \Theta(\lambda n. \ln (\text{fact } n)) \)
  - by (subst bigtheta-sym) (intro asymp-eqv-imp-bigtheta asymp-eqv-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 \( (\lambda n. \log 2 (\text{fact } n)) \in O(\lambda n. \text{count-wc-queries} \{ Rs n \} \text{ (sorter } n)) \)

**proof** (intro bigoI [where \( c = 1 \]) always-eventually allI, goal-cases)

- **case** \( 1 n \)
  - have \( \text{norm} (\log 2 (\text{fact } n)) = \log 2 (\text{fact} \ (\text{length } [0..<n])) \) by simp
  - also from sorter[of \( n \)] have \( \ldots \leq \text{real} \ (\text{count-wc-queries} \ { Rs n \} \text{ (sorter } n)) \)
  - using count-queries-ge[of \( 0..<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**