Comparison-based Sorting Algorithms

Manuel Eberl

October 11, 2017

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 []
by (rule sorted-wrt.intros)

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
  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)]) 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 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)
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-cone-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 ⊆ A sorted-wrt R xs
x ∈ set xs y ∈ set xs
shows index xs x < index xs y ↔ (y, x) ∉ 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 ⊆ A sorted-wrt R xs distinct xs
i < length xs j < length xs
shows (xs!i, xs!j) ∈ R ↔ i ≤ j
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 ∈ set xs xs!j ∈ set xs by (auto simp: set-conv-nth)
with assms(2) have xs!i ∈ A xs!j ∈ 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) ∈ R by (auto simp add: sorted-wrt-conv-nth)
moreover from assms and greater have xs!i ≠ xs!j by (simp add: nth-eq-iff-index-eq)
ultimately show ?thesis using linorder-on A R; greater by (auto simp: linorder-on-def antisym-def)
qed

1.4 Converting a list into a linear ordering

definition linorder-of-list :: ′a list ⇒ (′a × ′a) set where
linorder-of-list xs = {(a, b). a ∈ set xs ∧ b ∈ set xs ∧ index xs a ≤ index xs b}

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

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

1.5 Insertion sort

primrec insert-wrt :: (′a × ′a) set ⇒ ′a list ⇒ ′a list where
insert-wrt R [] = [x]
| insert-wrt R (y # ys) = (if (x, y) ∈ R then x # y # ys 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 (insort-wrt R xs)
proof
  from assms have set (insort-wrt R xs) = set xs ∧ sorted-wrt R (insort-wrt R xs)
    by (induction xs) (auto simp: insort-wrt-def intro: sorted-wrt-insert-wrt)
  thus ?thesis ..
qed

lemma distinct-insort-wrt [simp]: distinct (insort-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)
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 (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) = 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

7
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) by simp-all
  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)
    sorted-wrt R (sorted-wrt-list-of-set R B)
proof –
  have ∃!xs. set xs = B ∧ distinct xs ∧ sorted-wrt R xs
    (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

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** \(\text{linorder-rank} \) where

\[
\text{linorder-rank } R \ A \ x = \text{card } \{y \in A - \{x\}. \ (y,x) \in R\}
\]

**Lemma** \(\text{linorder-rank-le} \):

**Assumes** finite \(A\)

**Shows** \(\text{linorder-rank } R \ A \ x \leq \text{card } A\)

**Unfolding** \(\text{linorder-rank-def} \) using \(\text{assms}\)

by (rule \(\text{card-mono}\)) auto

**Lemma** \(\text{linorder-rank-less} \):

**Assumes** finite \(A\) \(x \in A\)

**Shows** \(\text{linorder-rank } R \ A \ x < \text{card } A\)

**Proof** –

have \(\text{linorder-rank } R \ A \ x \leq \text{card } (A - \{x\})\)

unfolding \(\text{linorder-rank-def} \) using \(\text{assms}\) by (rule \(\text{card-mono}\)) auto

also from \(\text{assms}\) have \(\ldots < \text{card } A\) by (intro \(\text{psubset-card-mono}\)) auto

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

**Qed**

**Lemma** \(\text{linorder-rank-union} \):

**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** –

have \(\text{linorder-rank } R \ (A \cup B) \ x = \text{card } \{y \in (A \cup B) - \{x\}. \ (y,x) \in R\}\)

by (simp add: \(\text{linorder-rank-def}\))

also have \(\{y \in (A \cup B) - \{x\}. \ (y,x) \in R\} = \{y \in A - \{x\}. \ (y,x) \in R\} \cup \{y \in B - \{x\}. \ (y,x) \in R\}\)

(by blast)

also have \(\text{card } \ldots = \text{linorder-rank } R \ A \ x + \text{linorder-rank } R \ B \ x\) unfolding \(\text{linorder-rank-def}\)

using \(\text{assms}\) by (intro \(\text{card-Un-disjoint}\)) auto

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

**Qed**

**Lemma** \(\text{linorder-rank-empty} \) [simp]: \(\text{linorder-rank } R \ \{\} \ x = 0\)

by (simp add: \(\text{linorder-rank-def}\))

**Lemma** \(\text{linorder-rank-singleton} \):

\(\text{linorder-rank } R \ \{y\} \ x = (\text{if } x \neq y \land (y,x) \in R \text{ then } 1 \text{ else } 0)\)

**Proof** –

have \(\text{linorder-rank } R \ \{y\} \ x = \text{card } \{z \in \{y\} - \{x\}. \ (z,x) \in R\}\) by (simp add: \(\text{linorder-rank-def}\))

also have \(\{z \in \{y\} - \{x\}. \ (z,x) \in R\} = (\text{if } x \neq y \land (y,x) \in R \text{ then } \{y\} \text{ else } \{\})\)

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 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 R 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 and antisym: antisym R by (simp-all add: linorder-on-def)
    from assms antisym show \{y \in A - \{x\}. (y, x) \in R\} \subseteq \{ya \in A - \{y\}. (ya, y) \in R\}
    ultimately have \{z \in A - \{y\}. (z, y) \in R\} \subseteq \{z \in A - \{x\}. (z,x) \in R\} by blast
    thus \(?\text{thesis}\) 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 \(?\text{thesis}\) by (auto simp: linorder-on-def refl-on-def)
    next
    case False
    from assms(1) have trans: trans R by (simp-all add: linorder-on-def)
    from assms have x \in B y \in B by auto

10
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

qed

lemma *linorder-rank-eq-iff*:
  assumes *linorder-on B R* finite \(A \subseteq A\) \(A \times A \subseteq A\)
  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 – 3)]
  *linorder-rank-mono*[OF *assms*(1 – 3)]
  \(5, 4\)
  have \((x, y) \in R \land (y, x) \notin R\) 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 \(xs \subseteq B\) sorted-wrt \(R\) \(xs \in\) \(set xs \) distinct \(xs\)
  shows *linorder-rank R (set xs)* \(x = index xs x\)
proof –
  define \(j\) where \(j = index xs x\)
  from *assms* have \(*j < length xs*\) 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 \(y \in set xs\) for \(y\)
    using *linorder-on-cases*[OF *assms*(1), of \(x, y\)] *assms* that by auto
  from *assms* have \({y \in set xs \times \{x\}, (y, x) \in R\} = \{y \in set xs - \{x\}. index xs y < index xs x\}\)
    by (auto simp: sorted-wrt-linorder-index-less-iff[OF *assms*(1 – 3)] dest: *)
  also have \(*\): \(y \in set xs. index xs y < j\) by (auto simp: j-def)
  also have \(*\): \((\lambda i. xs ! i) \langle i, i < j\rangle\)
    proof safe
      fix \(y\) assume \(*y \in set xs index xs y < j\)
        moreover from *this* and \(*j*\) have \(*y = xs ! index xs y*\) by simp
        ultimately show \(*y \in op ! xs \langle i, i < j\rangle*\) by blast
    qed (insert *assms* \(j, auto simp: index-nth-id\))
  also from *assms* and \(*j*\) have \(*\): \(y \in set xs. index xs y < j\)
    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
  by simp_all
from xs len-xs show ?thesis
  by (intro bij_betw_byWitness[where f' = λi. xs ! i]) (auto simp: rank index_nth_id intro: nth_mem)
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] 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
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).

```
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)))

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))

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

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

```
lemma card-range-eval-sorter: assumes finite Rs
  shows card ((λR. count-queries R sorter) ‘ Rs) ≤ 2 ^ count-wc-queries Rs sorter
  using assms
```

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

```
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.

```
lemma card-range-eval-sorter:
  assumes finite Rs
  shows card ((λR. eval-sorter R e) ‘ Rs) ≤ 2 ^ count-wc-queries Rs e
  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 f False ∈ 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)
  thus ?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-UnI finite-imageI fin *)
  also have ... ≤ card ?A + card ?B by (rule card-Un-le)
  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**  
\( \text{is-sorting} :: (\forall a \times a \text{ set} \Rightarrow \forall \text{ a list} \Rightarrow \forall \text{ a list} \Rightarrow \text{ bool}) \)  
where  
\( \text{is-sorting } R \ x s \ y s \leftarrow (\text{mset } x s = \text{mset } y s) \land \text{sorted-wrt } R \ y s \)

### 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**  
\[ \text{fixes } \text{sorter} :: \forall a \text{ sorter} \text{ and } \text{xs} :: \forall a \text{ list} \]  
\[ \text{assumes } \text{distinct}: \text{distinct } \text{xs} \]  
\[ \text{assumes } \text{sorter}: \land R. \text{linorder-on } (\text{set } \text{xs}) \Rightarrow \text{is-sorting } R \ \text{xs} \ (\text{eval-sorter } R \ \text{sorter}) \]  
\[ \text{defines } \text{Rs} \equiv \{ R. \text{linorder-on } (\text{set } \text{xs}) \} \]  
\[ \text{shows } \text{two-power-count-queries-ge}: \text{fact } (\text{length } \text{xs}) \leq (2 \ ^{\text{count-wc-queries } \text{Rs} \ \text{sorter}}) \]  
\[ \text{and } \text{count-queries-ge}: \text{log } 2 (\text{fact } (\text{length } \text{xs})) \leq \text{real } (\text{count-wc-queries } \text{Rs} \ \text{sorter}) \]  
**proof**  
\[ \text{have } \text{Rs} \subseteq \text{Pow } (\text{set } \text{xs} \times \text{set } \text{xs}) \text{ by } (\text{auto simp: Rs-def linorder-on-def refl-on-def}) \]  
\[ \text{hence } \text{fin: finite } \text{Rs by } (\text{rule finite-subset simp-all}) \]  
\[ \text{from } \text{assms have } \text{fact } (\text{length } \text{xs}) = \text{card } (\text{permutations-of-set } (\text{set } \text{xs})) \]  
\[ \text{by } (\text{simp add: distinct-card}) \]  
\[ \text{also have } \text{permutations-of-set } (\text{set } \text{xs}) \subseteq (\lambda R. \text{eval-sorter } R \ \text{sorter}) \ ^{\prime} \ \text{Rs} \]  
**proof**  
\[ \text{(rule subsetI, goal-cases)} \]  
\[ \text{case } (1 \ \text{ys}) \]  
\[ \text{define } \text{R where } \text{R} = \text{linorder-of-list } \text{ys} \]  
\[ \text{define } \text{zs where } \text{zs} = \text{eval-sorter } R \ \text{sorter} \]  
\[ \text{from } 1 \text{ and } \text{distinct have } \text{mset-ys: mset } \text{ys} = \text{mset } \text{xs} \]  
\[ \text{by } (\text{auto simp: set-eq-iff-mset-eq-distinct permutations-of-set-def}) \]  
\[ \text{from } 1 \text{ have } \ast: \text{linorder-on } (\text{set } \text{xs}) \ \text{R unfolding } \text{R-def using } \text{linorder-linorder-of-list[of ys]} \]  
\[ \text{by } (\text{simp add: permutations-of-set-def}) \]  
\[ \text{from } \text{sorter[(OF this)] have mset } \text{xs} = \text{mset } \text{zs } \text{sorted-wrt } \text{R } \text{zs} \]  
\[ \text{by } (\text{simp-all add: is-sorting-def zs-def}) \]  
\[ \text{moreover from } 1 \text{ have } \text{sorted-wrt } \text{R } \text{zs unfolding } \text{R-def} \]  
\[ \text{by } (\text{intro sorted-wrt-linorder-of-list}) \text{ (simp-all add: permutations-of-set-def}) \]  
\[ \text{ultimately have } \text{zs} = \text{ys} \]  
\[ \text{by } (\text{intro sorted-wrt-linorder-unique[OF } \ast]) \text{ (simp-all add: mset-ys}) \]  
\[ \text{moreover from } \ast \text{ have } \text{R} \in \text{Rs by } (\text{simp add: Rs-def}) \]  
\[ \text{ultimately show } \text{?case unfolding } \text{zs-def by blast} \]  
**qed**  
\[ \text{hence } \text{card } (\text{permutations-of-set } (\text{set } \text{xs})) \leq \text{card } (\lambda R. \text{eval-sorter } R \ \text{sorter}) \ ^{\prime} \ \text{Rs} \]  
\[ \text{by } (\text{intro card-mono finite-imageI fin}) \]
also from fin have \( \ldots \leq 2^n \cdot \text{count-wc-queries} \) Rs sorter by (rule card-range-eval-sorter)
finally show \( \ast \): \( \text{fact} (\text{length} \ xs) \leq (2^n \cdot \text{count-wc-queries} \) Rs sorter :: nat \).

have \( \ln (\text{fact} (\text{length} \ xs)) = \ln (\text{real} (\text{fact} (\text{length} \ xs))) \) by simp
also have \( \ldots \leq \ln (\text{real} (2^n \cdot \text{count-wc-queries} \) Rs sorter) \)
proof (subst ln-le-cancel-iff)
  
  show real (\text{fact} (\text{length} \ xs)) \leq real (2^n \cdot \text{count-wc-queries} \) Rs sorter
  by (subst of-nat-le-iff) (rule *)
qed simp-all

also have \( \ldots = \text{real} (\text{count-wc-queries} \) Rs sorter \) * ln 2 by (simp add: ln-rerealpow)
finally have \( \text{real} (\text{count-wc-queries} \) Rs sorter) \geq \ln (\text{fact} (\text{length} \ xs)) / ln 2
by (simp add: field-simps)
also have \( \ln (\text{fact} (\text{length} \ xs)) / ln 2 = \log 2 (\text{fact} (\text{length} \ xs)) \) by (simp add: log-def)
finally show \( \ast^\ast \): \( \log 2 (\text{fact} (\text{length} \ xs)) \leq \text{real} (\text{count-wc-queries} \) Rs sorter) .

qed


lemma \( \ln-fact-bigo \): \( (\lambda n. \ln (\text{fact} n) - (\ln (2 * \pi * n) / 2 + n * \ln n - n)) \in O(\ln n / n) \)
and \( \text{asympeq-ln-fact} \) \[ \text{asympeq-ln-intros]: (\lambda n. \ln (\text{fact} n)) \sim (\lambda n. n * \ln n) \]

proof
  
  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 \( (\lambda n. \ln (\text{fact} n) - f n \in \{0..1/(12*\text{real} n)\}) \) at-top
  using eventually-\( \ast \)gt-at-top[of 1::nat]
proof eventually-elim
  case (elim n)
  with ln-fact-bounds[of n] show ?case by (simp add: f-def)
qed

hence eventually \( (\lambda n. \text{norm} \ (\ln (\text{fact} n) - f n) \leq (1/12) * \text{norm} (1 / \text{real} n)) \)
at-top
using eventually-\( \ast \)gt-at-top[of 0::nat] by eventually-elim (simp-all add: field-simps)
thus \( (\lambda n. \ln (\text{fact} n) - f n \in O(\ln n / \text{real} n) \)
using bigo[of \( \lambda n. \ln (\text{fact} n) - f n \) / 12 \( \lambda n. n / \text{real} n \) by simp
also have \( (\lambda n. 1 / \text{real} n) \in o(f) \) unfolding f-def by (intro smallo-real-nat-transfer)
simp
finally have \( (\lambda n. f n + (\ln (\text{fact} n) - f n)) \sim f \)
by (subst \text{asympeq-equiv-add-right}) simp-all
hence \( (\lambda n. \ln (\text{fact} n)) \sim f \) by simp
also have \( f \sim (\lambda n. n * \ln n + (\ln (2 * \pi * n) / 2 - n)) \) by (simp add: f-def algebra-simps)
also have \( \ldots \sim (\lambda n. n * \ln n) \) by (subst \text{asympeq-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 \text{count-queries-bigomega}:
fixes sorter :: nat ⇒ nat sorter
assumes sorter: ⋀ n. linorder-on {..<n} R ⇒
        is-sorting R [0..<n] (eval-sorter R (sorter n))
defines Rs ≡ λ n. { R. linorder-on {..<n} R }
shows (λ n. count-wc-queries (Rs n) (sorter n)) ∈ Ω(λ n. n * ln n)
proof –
  have (λ n. n * ln n) ∈ Θ(λ n. ln (fact n))
     by (subst bigtheta-sym) (intro asymp-eqv-imp-bigtheta asymp-eqv-intros)
also have (λ n. ln (fact n)) ∈ Θ(λ n. log 2 (fact n)) by (simp add: log-def)
also have (λ n. log 2 (fact n)) ∈ O(λ n. count-wc-queries (Rs n) (sorter n))
proof (intro bigoI[where c = 1] always-eventually allI, goal-cases)
  case (Suc n)
  have (λ n. ln (fact n)) = log 2 (fact (length [0..<n])) by simp
  also have ... ≤ real (count-wc-queries (Rs n) (sorter n))
  using count-queries-ge[of [0..<n] sorter n] by (auto simp: Rs-def atLeast0LessThan)
also have ... = 1 * norm ... by simp
finally show ?case by simp
qed
finally show ?thesis by (simp add: bigomega-iff-bigo)
qed
end

References