Comparison-based Sorting Algorithms

Manuel Eberl

April 19, 2020

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
∧ (∀ x y. x ∈ set xs ⇒ y ∈ set ys ⇒ (x,y) ∈ R 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 (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)])/2(2)(OF */[])) simp
  ultimately show ?thesis using 2 False
  by (auto intro!: sorted-wrt.intros)
qed (insert 2, auto intro: sorted-wrt.intros)

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 -
define i j where i = index xs x and j = index xs y
{
  assume j < i
  moreover from assms have i < length xs by (simp add: i-def)
  ultimately have (xs!i..xs!j) ∈ R using assms by (auto simp: sorted-wrt-cone-nth)
  with assms have x = y by (auto simp: linorder-on-def antisym-def i-def j-def)
}
hence i ≤ j ∨ x = y by linarith
thus ?thesis by (auto simp: 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-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 $A R \subseteq A$ sorted-wrt $R \subseteq A$

$x \in \text{set } xs$ $y \in \text{set } xs$

shows $\text{index } xs x < \text{index } xs y \iff (y, x) \notin R$

by (subst sorted-wrt-linorder-index-le-iff \begin{itemize}
    \item \(\text{OF assms(1-3)}\)
    \item \(\text{assms(5,4), symmetric}\)
\end{itemize}) auto

lemma sorted-wrt-distinct-linorder-nth:
assumes linorder-on $A R \subseteq A$ sorted-wrt $R \subseteq A$ distinct $xs$

$i < \text{length } xs$ $j < \text{length } xs$

shows $\langle x \text{!} i, x \text{!} j \rangle \in R \iff i \leq j$

proof (cases $i$ $j$ rule: linorder-cases)

next

with \text{assms} show $?\text{thesis}$ by (simp add: sorted-wrt-conv-nth)

next

case equal

from \text{assms} and equal have $x \text{!} i \in \text{set } xs$ $x \text{!} j \in \text{set } xs$

by (auto simp: set-conv-nth)

with \text{assms(2)} have $x \text{!} i \in A$ $x \text{!} j \in A$

by blast+ 

with \langle linorder-on $A R$ \rangle and equal show $?\text{thesis}$ by (simp add: linorder-on-def refl-on-def)

next

case greater

with \text{assms} have $\langle x \text{!} j, x \text{!} i \rangle \in R$

by (auto simp add: sorted-wrt-conv-nth)

moreover from \text{assms and greater} have $x \text{!} i \neq x \text{!} j$

by (simp add: nth-eq-iff-index-eq)

ultimately show $?\text{thesis}$ using \langle linorder-on $A R$ \rangle 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 $\Rightarrow$ $'a$ $\times$ $'a$ set where

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

insert-wrt $R$ $x$ $\langle \rangle$ = $[x]$

| insert-wrt $R$ $x$ $\langle y \# ys \rangle$ = (if $(x, y) \in R$ then $x \# y \# ys$ else $y \# insert-wrt$ $R$ $x$ $ys$)

lemma set-insert-wrt [simp]: $\text{set } (\text{insert-wrt } R x xs) = \text{insert } x \text{ (set } xs)$
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]:
\quad linorder-on A R ⇒ set (x ∈ A) ⊆ A ⇒
\quad sorted-wrt R xs ⇒ sorted-wrt R (insert-wrt R x xs)

proof (induction xs)
\quad case (Cons y ys)
\quad from Cons.prems have (x, y) ∈ R ∨ (y, x) ∈ R
\quad by (cases x = y) (auto simp: linorder-on-def refl-on-def total-on-def)
\quad with Cons show ?case
\quad by (auto simp: sorted-wrt-Cons intro: transD simp: linorder-on-def)
qed auto

lemma sorted-wrt-insort [intro]:
\quad assumes linorder-on A R set xs ⊆ A
\quad shows sorted-wrt R (insert-wrt R xs)

proof –
\quad from assms have set (insert-wrt R xs) = set xs ∧ sorted-wrt R (insert-wrt R xs)
\quad by (induction xs) (auto simp: insert-wrt-def intro!: sorted-wrt-insert-wrt)
\quad 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:
\quad assumes linorder-on A R mset xs = mset ys sorted-wrt R xs sorted-wrt R ys
\quad shows xs = ys

proof –
\quad 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
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)
    using zsa assms (1)
  finally show ?thesis using zsa assms
    by (intro sorted-wrt-linorder-unique[OF assms(1)])
    (auto intro!: sorted-wrt-insort)

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
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 \( \{ y \in A - \{ x \} . (y,x) \in R \} \)

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

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

lemma linorder-rank-union:
assumes finite \( A \) finite \( B \) \( A \cap B = \{ \} \)
shows linorder-rank \( R \) \( (A \cup B) \ x \) = linorder-rank \( R \ A \ x \) + linorder-rank \( R \ B \ x \)
proof
have linorder-rank \( R \) \( (A \cup B) \ x \) = card \( \{ y \in (A \cup B) - \{ x \} . (y,x) \in R \} \)
by (simp add: 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 card \( \ldots = \) 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 \( R \) \( \{ y \} \ x = (if x \neq y \land (y,x) \in R \ then \ 1 \ else \ 0) \)
proof
have linorder-rank \( R \) \( \{ y \} \ x = card \{ z \in \{ y \} - \{ x \} \ . (z,x) \in R \} \)
by (simp add: linorder-rank-def)
also have \( \{ z \in \{ y \} - \{ x \} \ . (z,x) \in R \} = (if x \neq y \land (y,x) \in R \ then \ \{ y \} \ else \ \{ \} ) \)
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

definition 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)

definition 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 \( \{y \in A - \{x\}. (y, x) \in R\} \subseteq \{ya \in A - \{y\}. (ya, y) \in R\} \)
qed (insert assms, simp-all)

definition linorder-rank-strict-mono:
assumes linorder-on B R finite A y \in A \ (y, x) \in R x \neq y
shows linorder-rank R A y < linorder-rank R A x
proof –
  from assms(1) have trans: trans R by (simp add: linorder-on-def)
  from assms have *: \( (x, y) \notin R \) by (auto simp: linorder-on-def antisym-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])
  moreover from * and assms have \( y \notin \{z \in A - \{y\}. (z, y) \in R\} \) \( y \in \{z \in A - \{x\}. (z, x) \in R\} \)
  by auto
  ultimately have \( \{z \in A - \{y\}. (z, y) \in R\} \subset \{z \in A - \{x\}. (z, x) \in R\} \) by blast
  thus ?thesis using assms unfolding linorder-rank-def by (intro psubset-card-mono)
auto
qed

definition 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-all 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 \(x \in\) A y \(\in\) A

shows \(\text{linorder-rank} R A x = \text{linorder-rank} R A y \iff x = y\)

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)]

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 x \(\in\) set xs distinct xs

shows \(\text{linorder-rank} R (\text{set} xs) x = \text{index} xs x\)

proof

- define \(j\) where \(j = \text{index} xs x\)

from assms have \(j < \text{length} xs\) 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 \(y \in \text{set} xs\) for \(y\)

using linorder-on-cases[OF assms(1), of x y] assms that by auto

from assms have \(\{y \in \text{set} xs \mid (x, y) \in R\} = \{y \in \text{set} xs \mid (x, y) \notin R\}\)

by (auto simp: sorted-wrt-linorder-index-less-iff[OF assms(1-3)] dest: *)

also have \(\ldots = \{y \in \text{set} xs. \text{index} xs y < j\}\) by (auto simp: j-def)

also have \(\ldots = (\lambda i. \text{xs ! i} \cdot \{i. i < j\})\)

proof safe

fix \(y\) assume \(y \in \text{set} xs\) \(\text{index} xs y < j\)

moreover from \textit{this} and \(j\) have \(y = \text{xs ! index xs y}\) by simp

ultimately show \(y \in (!) \text{xs} \cdot \{i. i < j\}\) by blast

qed (insert assms j, auto simp: index-nth-id)

also from assms and \(j\) have \(\text{card} \ldots = \text{card} \{i. i < j\}\)

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] assms that xs
  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,
        goal-cases)
case (1 xs)
thus ?case by (auto 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).

\[\text{primrec count-queries} :: (\text{'a} \times \text{'a}) \text{ set} \Rightarrow \text{'a sorter} \Rightarrow \text{nat} \]
\[
\text{where} \quad \text{count-queries} - (\text{Return} -) = 0 \\
\quad \text{count-queries} R (\text{Query} a b f) = \text{Suc} (\text{count-queries} R (f ((a, b) \in R)))
\]

\[\text{definition count-wc-queries} :: (\text{'a} \times \text{'a}) \text{ set set} \Rightarrow \text{'a sorter} \Rightarrow \text{nat} \]
\[
\text{where} \quad \text{count-wc-queries} Rs sorter = (\text{if} Rs = \{} \text{ then} 0 \text{ else} \text{Max} ((\lambda R. \text{count-queries} R \text{ sorter}) \ ' Rs))
\]

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

\[\text{lemma count-wc-queries-aux:}
\begin{align*}
\text{assumes } & (\forall R. R \in Rs \Rightarrow \text{sorter} = \text{sorter}' \ R \subseteq Rs' \ \text{finite Rs'}) \\
\text{shows } & \text{count-wc-queries} Rs \text{ sorter} \leq \text{Max} (\lambda R. \text{count-queries} R \text{ sorter'} R) \ ' Rs')
\end{align*}
\]
\[
\text{proof (cases Rs = {}})
\begin{align*}
\text{case False} \\
& \text{hence count-wc-queries} Rs \text{ sorter} = \text{Max} (\lambda R. \text{count-queries} R \text{ sorter} \ ' Rs) \\
& \text{by (simp add: count-wc-queries-def)}
\end{align*}
\[
\text{also have} \ (\lambda R. \text{count-queries} R \text{ sorter}) \ ' Rs = (\lambda R. \text{count-queries} R \text{ sorter'} R) \ ' Rs \\
\text{by (intro image-cong refl) (simp-all add: assms)}
\]
\[
\text{also have Max . . . \leq Max} (\lambda R. \text{count-queries} R \text{ sorter'} R) \ ' Rs') \text{ using False}
\]
\[
\text{by (intro Max-mono assms image-mono finite-imageI) auto}
\]
\[
\text{finally show} \ ?\text{thesis} .
\]
\[
\text{qed simp-all}
\]

\[\text{primrec eval-sorter} :: (\text{'a} \times \text{'a}) \text{ set} \Rightarrow \text{'a sorter} \Rightarrow \text{'a list} \]
\[
\text{where} \quad \text{eval-sorter} - (\text{Return} ys) = ys \\
\quad \text{eval-sorter} R (\text{Query} a b f) = \text{eval-sorter} R (f ((a, b) \in R))
\]

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

\[\text{lemma card-range-eval-sorter:}
\begin{align*}
\text{assumes } & \text{finite Rs} \\
\text{shows } & \text{card} ((\lambda R. \text{eval-sorter} R \text{ e}) \ ' Rs) \leq 2 \ ^{\text{count-wc-queries} Rs \text{ e}}
\end{align*}
\]
The following predicate describes what constitutes a valid sorting result for

using \textit{assms}

\textbf{proof (induction e arbitrary: Rs)}

\textbf{case (Return $xs$ Rs)}

\textbf{have \textit{unfolding}}: $(\lambda R. \text{eval-sorter } R (\text{Return } xs))' \cdot Rs = (\text{if } Rs = \{\} \text{ then } \{\} \text{ else } \{xs\})$

\textbf{by auto}

\textbf{show} ?\textit{case} \textbf{by} (subst \textit{unfolding}) \textbf{auto}

\textbf{next}

\textbf{case (Query \textit{a} \textit{b} \textit{f} \textit{Rs})}

\textbf{have \textit{f True} $\in$ range \textit{f} $\text{False} \in$ range \textit{f} \textbf{by} simp-all

\textbf{note} \textit{IH} = this \textbf{[THEN Query.IH]}

\textbf{let} ?Rs1 = \{\textit{R}$\in$Rs. (\textit{a}, \textit{b}) $\in$ \textit{R}\} \textbf{and} ?Rs2 = \{\textit{R}$\in$Rs. (\textit{a}, \textit{b}) $\notin$ \textit{R}\}

\textbf{let} ?\textit{A} = $(\lambda R. \text{eval-sorter } R (f \text{ True}))' \cdot \text{ ?Rs1 and } ?B = (\lambda R. \text{eval-sorter } R (f \text{ False}))' \cdot \text{ ?Rs2}

\textbf{from Query.prems have} \textit{fin: finite} \textit{?Rs1 finite ?Rs2 by simp-all

\textbf{have \textit{unfolding}}: $(\lambda R. \text{eval-sorter } R (\text{Query } \textit{a} \textit{b} \textit{f} ))' \cdot \textit{Rs} \subseteq \textit{?A} \cup \textit{?B}$

\textbf{proof} (intro subsetI, elim imageE, goal-cases)

\textbf{case (1 \textit{xs} \textit{R})}

\textbf{thus} ?\textit{case} \textbf{by} (cases (\textit{a},\textit{b}) $\in$ \textit{R}) \textbf{auto

\textbf{qed

\textbf{show} ?\textit{case}

\textbf{proof} (cases \textit{Rs} = \{\})

\textbf{case False}

\textbf{have} \textit{card} $(\lambda R. \text{eval-sorter } R (\text{Query } \textit{a} \textit{b} \textit{f} ))' \cdot \textit{Rs} \leq \text{ card } \textit{?A} \cup \textit{?B}$

\textbf{by} (intro card_mono finite-UnI finite-imageI fin \textit{unfolding \textit{IH}})

\textbf{also have} \ldots $\leq$ \textit{card} \textit{?A}$+$ \textit{card} ?\textit{B} \textbf{by} (rule card-Un-le)

\textbf{also have} \ldots $\leq 2 ^\ell$ \textit{count-wc-queries} ?\textit{Rs1} (\textit{f True})$+$ \textit{2 ^\ell$ \textit{count-wc-queries} ?\textit{Rs2} (\textit{f False})$)$

\textbf{by} (intro add_mono \textit{IH fin})

\textbf{also have} \textit{count-wc-queries} ?\textit{Rs1} (\textit{f True}) $\leq$ \textit{Max} $(\lambda R. \text{count-queries } R (f ((\textit{a},\textit{b})\in\textit{R})))' \cdot \textit{Rs})$

\textbf{by} (intro count-wc-queries-aux Query.prems) \textbf{auto

\textbf{also have} \textit{count-wc-queries} ?\textit{Rs2} (\textit{f False}) $\leq$ \textit{Max} $(\lambda R. \text{count-queries } R (f ((\textit{a},\textit{b})\in\textit{R})))' \cdot \textit{Rs})$

\textbf{by} (intro count-wc-queries-aux Query.prems) \textbf{auto

\textbf{also have} \textit{2 ^\ell$ \ldots $+$ \textit{2 ^\ell$ \ldots $ = (2 ^\ell$ \textit{Suc} \ldots $:$ nat) by \textit{simp

\textbf{also have} \textit{Suc} (\textit{Max} $(\lambda R. \text{count-queries } R (f ((\textit{a},\textit{b})\in\textit{R})))' \cdot \textit{Rs})$) = \textit{Max} (\textit{Suc} $(\lambda R. \text{count-queries } R (f ((\textit{a},\textit{b})\in\textit{R})))' \cdot \textit{Rs})$ $\textit{using False}

\textbf{by} (intro mono-Max-commute finite-imageI Query.prems) (auto simp: incseq-def)

\textbf{also have} \textit{Suc} $(\lambda R. \text{Suc} (\text{count-queries } R (f ((\textit{a},\textit{b})\in\textit{R})))' \cdot \textit{Rs})$ $\textit{by} (\textit{simp add: image-image})$

\textbf{also have} \textit{Max} \ldots $\textit{= count-wc-queries} \textit{Rs} (\text{Query } \textit{a} \textit{b} \textit{f}) \textit{using False}

\textbf{by} (auto simp add: count-wc-queries-def)

\textbf{finally show} ?\textit{thesis} \textbf{by} = \textit{simp-all

\textbf{qed simp-all

\textbf{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: \( \forall R. \text{linorder-on (set xs)}\) \( R \implies \text{is-sorting R xs (eval-sorter R sorter)} \)
  defines Rs ≡ \( \{ R. \text{linorder-on (set xs)} R \} \)
  shows two-power-count-queries-ge: \( \text{fact (length xs)} \leq (2^\text{count-wc-queries Rs sorter}) \)
    and count-queries-ge: \( \log 2 (\text{fact (length xs)}) \leq \text{real (count-wc-queries Rs sorter)} \)
  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 fact (length xs) = card (permutations-of-set (set xs))
      by (simp add: distinct-card)
    also have permutations-of-set (set xs) ⊆ (\( \lambda R. \text{eval-sorter R sorter} \)) ▶ Rs
      proof (rule subsetI, goal_cases)
        case (1 ys)
        define R where R = linorder-of-list ys
        define zs where zs = 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 *: linorder-on (set xs) R unfolding R-def using linorder-linorder-of-list[of ys]
          by (simp add: permutations-of-set-def)
        from sorter[OF this] have mset zs = mset zs sorted-wrt R zs
          by (simp-all add: is-sorting-def zs-def)
        moreover from 1 have sorted-wrt R ys unfolding R-def
          by (intro sorted-wrt-linorder-of-list) (simp-all add: permutations-of-set-def)
        ultimately have zs = ys
          by (intro sorted-wrt-linorder-unique[OF *]) (simp-all add: mset-ys)
        moreover from * have R ∈ Rs by (simp add: Rs-def)
        ultimately show ?case unfolding zs-def by blast
      qed
    hence card (permutations-of-set (set xs)) ≤ card ((\( \lambda R. \text{eval-sorter R sorter} \)) ▶ Rs)
      by (intro card-mono finite-imageI fin)
also from fin have ... ≤ 2 ^ count-wc-queries Rs sorter by (rule card-range-eval-sorter)
finally show *: fact (length xs) ≤ (2 ^ count-wc-queries Rs sorter :: nat).

have ln (fact (length xs)) = ln (real (fact (length xs))) by simp
also have ... ≤ ln (real (2 ^ count-wc-queries Rs sorter))
proof (subst ln-le-cancel-iff)
  show real (fact (length xs)) ≤ real (2 ^ count-wc-queries Rs sorter)
    by (subst of-nat-le-iff) (rule *)
qed simp-all
also have ... = real (count-wc-queries Rs sorter) * ln 2 by (simp add: ln-realpow)
finally have real (count-wc-queries Rs sorter) ≥ ln (fact (length xs)) / ln 2
  by (simp add: field-simps)
also have ln (fact (length xs)) / ln 2 = log 2 (fact (length xs)) by (simp add: log-def)
finally show **: log 2 (fact (length xs)) ≤ real (count-wc-queries Rs sorter).

lemma ln-fact-bigo: (λn. ln (fact n) − (ln (2 * pi * n) / 2 + n * ln n − n)) ∈ O(λn. 1 / n)
  and asymp-equn-fact [asymp-equn-intros]; (λn. ln (fact n)) ∼[at-top] (λn. n * ln n)
proof —
  include asymp-equn-notation
define f where f = (λn. ln (2 * pi * real n) / 2 + real n * ln (real n) − real n)
have eventually (λn. ln (fact n) − f n ∈ {0..1/(12*real n)}) at-top
  using eventually-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 (λn. norm (ln (fact n) − f n) ≤ (1/12) * norm (1 / real n))
  at-top
  using eventually-gt-at-top[of 0::nat] by eventually-elim (simp-all add: field-simps)
thus (λn. ln (fact n) − f n) ∈ O(λn. 1 / real n)
  using bigo[of λn. ln (fact n) − f n 1/12 λn. 1 / real n] by simp
also have (λn. 1 / real n) ∈ o(f) unfolding f-def by (intro smallo-real-nat-transfer)
simp
finally have (λn. f n + (ln (fact n) − f n)) ∼ f
  by (subst asymp-equn-add-right) simp-all
hence (λn. ln (fact n)) ∼ f by simp
also have f ∼ (λn. n * ln n + (ln (2*pi*n)/2 − n)) by (simp add: f-def algebra-simps)
also have ... ∼ (λn. n * ln n) by (subst asymp-equn-add-right) auto
finally show (λn. ln (fact n)) ∼ (λn. n * ln n).

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: ∀n. linorder-on {..<n} R ⇒
is-sorting R [0..<n] (eval-sorter R (sorter n))
defines Rs ≡ λn. {R. linorder-on {..<n} R}
shows (λn. count-wc-queries (Rs n) (sorter n)) ∈ Ω(λn. n * ln n)

proof –
  have (λn. n * ln n) ∈ Θ(λn. ln (fact n))
    by (subst bigtheta-sym) (intro asymp-eqv-imp-bigtheta asymp-eqv-intros)
  also have (λn. ln (fact n)) ∈ Θ(λn. log 2 (fact n)) by (simp add: log-def)
  also have (λn. log 2 (fact n)) ∈ O(λn. count-wc-queries (Rs n) (sorter n))
  proof (intro bigoI [where c = 1] always-eventually allI, goal-cases)
    case (1 n)
    have norm (log 2 (fact n)) = log 2 (fact (length [0..<n])) by simp
    also from sorter[of n] 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