Abstract

This article contains a formal proof of the well-known fact that 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 .......................................... 3
  1.4 Converting a list into a linear ordering ................ 4
  1.5 Insertion sort ............................................ 4
  1.6 Obtaining a sorted list of a given set .................. 5
  1.7 Rank of an element in an ordering ....................... 6
  1.8 The bijection between linear orderings and lists .... 7

2 Lower bound on costs of comparison-based sorting 7
  2.1 Abstract description of sorting algorithms ............. 7
  2.2 Lower bounds on number of comparisons ................ 9
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⟩

lemma distinct-mset-mono:
  assumes distinct ys mset xs ⊆# mset ys
  shows  distinct xs
⟨proof⟩

lemma mset-eq-imp-distinct-iff:
  assumes mset xs = mset ys
  shows  distinct xs ←→ distinct ys
⟨proof⟩

lemma total-on-subset: total-on B R ⇒ A ⊆ B ⇒ total-on A R
⟨proof⟩

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 ∈ set xs ⇒ (x,y) ∈ R) ⇒ sorted-wrt R (x # xs)

lemma sorted-wrt-Cons [simp]: sorted-wrt R []
⟨proof⟩

lemma sorted-wrt-Cons: sorted-wrt R (x # xs) ←→ (∀ y∈set xs. (x,y) ∈ R) ∧
sorted-wrt R xs
⟨proof⟩

lemma sorted-wrt-singleton [simp]: sorted-wrt R [x]
⟨proof⟩

lemma sorted-wrt-many:
  assumes trans R
  shows  sorted-wrt R (x ≠ y ≠ xs) ←→ (x,y) ∈ R ∧ sorted-wrt R (y ≠ xs)
⟨proof⟩

lemma sorted-wrt-imp-le-last:
assumes \( \text{sorted-wrt } R \) \( xs \) \( \neq \emptyset \) \( x \in \text{set } xs \) \( x \neq \text{last } xs \)
shows \((x, \text{last } xs) \in R\)
⟨proof⟩

lemma \text{sorted-wrt-append}:
assumes \( \text{sorted-wrt } R \) \( xs \) \( \text{sorted-wrt } R \) \( ys \)
\( \forall x \ y. \ x \in \text{set } xs \implies y \in \text{set } ys \implies (x ,y) \in R \text{ trans } R \)
shows \( \text{sorted-wrt } R \) \( (xs @ ys) \)
⟨proof⟩

lemma \text{sorted-wrt-snoc}:
assumes \( \text{sorted-wrt } R \) \( xs \) \( (\text{last } xs, y) \in R \text{ trans } R \)
shows \( \text{sorted-wrt } R \) \( (xs @ [y]) \)
⟨proof⟩

lemma \text{sorted-wrt-conv-nth}:
\( \text{sorted-wrt } R \) \( xs \) \( \iff \) \( \forall i \ j . \ i < j \land j < \text{length } xs \implies (xs[i], xs[j]) \in R \)
⟨proof⟩

1.3 Linear orderings

definition \text{linorder-on} :: \('a set \Rightarrow ('a \times 'a) set \Rightarrow \text{bool} \) \where
\text{linorder-on } A \ R \iff \text{refl-on } A \ R \land \text{antisym } R \land \text{trans } R \land \text{total-on } A \ R

lemma \text{linorder-on-cases}:
assumes \( \text{linorder-on } A \ R \) \( x \in A \) \( y \in A \)
shows \( x = y \lor ((x ,y) \in R \land (y ,x) \notin R) \lor ((y ,x) \in R \land (x ,y) \notin R) \)
⟨proof⟩

lemma \text{sorted-wrt-linorder-imp-index-le}:
assumes \( \text{linorder-on } A \ R \) \( \text{set } xs \subseteq A \) \( \text{sorted-wrt } R \) \( xs \)
\( x \in \text{set } xs \) \( y \in \text{set } xs \) \( (x,y) \in R \)
shows \( \text{index } xs \leq \text{index } xs \ y \)
⟨proof⟩

lemma \text{sorted-wrt-linorder-index-le-imp}:
assumes \( \text{linorder-on } A \ R \) \( \text{set } xs \subseteq A \) \( \text{sorted-wrt } R \) \( xs \)
\( x \in \text{set } xs \) \( y \in \text{set } xs \) \( \text{index } xs \ x \leq \text{index } xs \ y \)
shows \((x,y) \in R\)
⟨proof⟩

lemma \text{sorted-wrt-linorder-index-le-iff}:
assumes \( \text{linorder-on } A \ R \) \( \text{set } xs \subseteq A \) \( \text{sorted-wrt } R \) \( xs \)
\( x \in \text{set } xs \) \( y \in \text{set } xs \)
shows \( \text{index } xs \ x \leq \text{index } xs \ y \iff (x,y) \in R \)
⟨proof⟩

lemma \text{sorted-wrt-linorder-index-less-iff}:
assumes \( \text{linorder-on } A \ R \) \( \text{set } xs \subseteq A \) \( \text{sorted-wrt } R \) \( xs \)

3
\[ x \in \text{set } xs \quad y \in \text{set } xs \]
\[ \text{shows} \quad \text{index } xs \ x < \text{index } xs \ y \iff (y, x) \notin R \]
\[ (\text{proof}) \]

**Lemma** sorted-wrt-distinct-linorder-nth:
\[
\text{assumes} \quad \text{linorder-on } A \ R \ \text{set } xs \subseteq A \ \text{sorted-wrt } R \ xs \ \text{distinct } xs \\
\text{i < length } xs \ \text{j < length } xs \\
\text{shows} \quad (xs!i, xs!j) \in R \iff i \leq j 
\]
\[ (\text{proof}) \]

### 1.4 Converting a list into a linear ordering

**Definition** linorder-of-list :: 'a list ⇒ ('a × 'a) set where
\[ \text{linorder-of-list } xs = \{(a, b). \ a \in \text{set } xs \land b \in \text{set } xs \land \text{index } xs \ a \leq \text{index } xs \ b\} \]

**Lemma** linorder-linorder-of-list [intro, simp]:
\[
\text{assumes} \quad \text{distinct } xs \\
\text{shows} \quad \text{linorder-on } (\text{set } xs) \ (\text{linorder-of-list } xs) 
\]
\[ (\text{proof}) \]

**Lemma** sorted-wrt-linorder-of-list [intro, simp]:
\[
\text{distinct } xs \implies \text{sorted-wrt } (\text{linorder-of-list } xs) \ xs 
\]
\[ (\text{proof}) \]

### 1.5 Insertion sort

**Primrec** insert-wrt :: ('a × 'a) set ⇒ 'a ⇒ 'a list ⇒ 'a 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** set-insert-wrt [simp]: \[ \text{set } (\text{insert-wrt } R \ x \ xs) = \text{insert } x \ (\text{set } xs) \]
\[ (\text{proof}) \]

**Lemma** mset-insert-wrt [simp]: \[ \text{mset } (\text{insert-wrt } R \ x \ xs) = \text{add-mset } x \ (\text{mset } xs) \]
\[ (\text{proof}) \]

**Lemma** length-insert-wrt [simp]: \[ \text{length } (\text{insert-wrt } R \ x \ xs) = \text{Suc } (\text{length } xs) \]
\[ (\text{proof}) \]

**Definition** insort-wrt :: ('a × 'a) set ⇒ 'a list ⇒ 'a list where
\[ \text{insort-wrt } R \ xs = \text{foldr } (\text{insert-wrt } R) \ xs [] \]

**Lemma** set-insort-wrt [simp]: \[ \text{set } (\text{insort-wrt } R \ xs) = \text{set } xs \]
\[ (\text{proof}) \]

**Lemma** mset-insort-wrt [simp]: \[ \text{mset } (\text{insort-wrt } R \ xs) = \text{mset } xs \]
\[ (\text{proof}) \]

**Lemma** length-insort-wrt [simp]: \[ \text{length } (\text{insort-wrt } R \ xs) = \text{length } xs \]
\[ (\text{proof}) \]
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⟩

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

lemma distinct-insort-wrt [simp]: distinct (insert-wrt R xs) ←→ distinct xs
⟨proof⟩

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⟩

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⟩

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⟩

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⟩

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⟩

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⟩
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**

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

**lemma linorder-rank-le:**

- **assumes** finite $A$
- **shows** $\text{linorder-rank } R \ A \ x \leq \text{card } A$

(proof)

**lemma linorder-rank-less:**

- **assumes** finite $A$ $x \in A$
- **shows** $\text{linorder-rank } R \ A \ x < \text{card } A$

(proof)

**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)

**lemma linorder-rank-empty [simp]:** $\text{linorder-rank } R \ \{ \} \ x = 0$

(proof)

**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)

**lemma linorder-rank-insert:**

- **assumes** finite $A$ $y \not\in A$
- **shows** $\text{linorder-rank } R (\text{insert } y \ A) \ x =\$

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

(proof)

**lemma linorder-rank-mono:**

- **assumes** linorder-on $B \ R$ finite $A \ A \subseteq B \ (x, y) \in R$
- **shows** $\text{linorder-rank } R \ A \ x \leq \text{linorder-rank } R \ A \ y$

(proof)

**lemma linorder-rank-strict-mono:**

- **assumes** linorder-on $B \ R$ finite $A \ A \subseteq B \ y \in A \ (y, x) \in R \ x \neq y$
- **shows** $\text{linorder-rank } R \ A \ y < \text{linorder-rank } R \ A \ x$

(proof)

**lemma linorder-rank-le-iff:**

- **assumes** linorder-on $B \ R$ finite $A \ A \subseteq B \ x \in A \ y \in A$

(proof)
shows \( \text{linorder-rank} \ R \ A \ x \leq \ \text{linorder-rank} \ R \ A \ y \iff (x, y) \in R \)

(\text{proof})

\begin{aligned}
\text{lemma linorder-rank-eq-iff:} \\
\text{assumes linorder-on} \ B \ R \ \text{finite} \ A \quad A \subseteq B \ x \in A \ y \in A \\
\text{shows linorder-rank} \ R \ A \ x = \ \text{linorder-rank} \ R \ A \ y \iff x = y \\
\end{aligned}

(\text{proof})

\begin{aligned}
\text{lemma linorder-rank-set-sorted-wrt:} \\
\text{assumes linorder-on} \ B \ R \ \text{set} \ \text{xs} \subseteq B \ \text{sorted-wrt} \ R \ \text{xs} \ x \in \text{set} \ \text{xs} \ \text{distinct} \ \text{xs} \\
\text{shows linorder-rank} \ R \ (\text{set} \ \text{xs}) \ x = \ \text{index} \ \text{xs} \ x \\
\end{aligned}

(\text{proof})

\begin{aligned}
\text{lemma bij-betw-linorder-rank:} \\
\text{assumes linorder-on} \ B \ R \ \text{finite} \ A \ A \subseteq B \\
\text{shows bij-betw} \ (\text{linorder-rank} \ R \ A) \ A \ \{..<\ \text{card} \ A\} \\
\end{aligned}

(\text{proof})

\[ \text{1.8 The bijection between linear orderings and lists} \]

\begin{aligned}
\text{theorem bij-betw-linorder-of-list:} \\
\text{assumes finite} \ A \\
\text{shows bij-betw linorder-of-list} \ (\text{permutations-of-set} \ A) \ \{R. \ \text{linorder-on} \ A \ R\} \\
\end{aligned}

(\text{proof})

\begin{aligned}
\text{corollary card-finite-linorders:} \\
\text{assumes finite} \ A \\
\text{shows card} \ \{R. \ \text{linorder-on} \ A \ R\} = \ \text{fact} \ (\text{card} \ A) \\
\end{aligned}

(\text{proof})

end

\[ \text{2 Lower bound on costs of comparison-based sorting} \]

\begin{aligned}
\text{theory Comparison-Sort-Lower-Bound} \\
\text{imports} \\
\quad \text{Complex-Main} \\
\quad \text{Linorder-Relations} \\
\quad \text{Stirling-Formula.Stirling-Formula} \\
\quad \text{Landau-Symbols.Landau-More} \\
\text{begin} \\
\end{aligned}

\[ \text{2.1 Abstract description of sorting algorithms} \]

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.

\[
\text{datatype } 'a \text{ sorter} = \text{Return } 'a \text{ list} | \text{Query } 'a 'a \text{ bool } \Rightarrow 'a \text{ sorter}
\]

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).

\[
\text{primrec } \text{count-queries} :: (a \times a) \text{ set } \Rightarrow 'a \text{ sorter } \Rightarrow \text{nat where}
\]

\[
\text{count-queries } (\text{Return } -) = 0
\]

\[
\text{count-queries } R (\text{Query } a b f) = \text{Suc} (\text{count-queries } f ((a, b) \in R))
\]

\[
\text{definition } \text{count-wc-queries} :: (a \times a) \text{ set set } \Rightarrow 'a \text{ sorter } \Rightarrow \text{nat where}
\]

\[
\text{count-wc-queries } Rs sorter = (\text{if } Rs = \{\} \text{ then } 0 \text{ else } \text{Max} (\lambda R. \text{count-queries } R \text{ sorter } ' RRs))
\]

\[
\text{lemma } \text{count-wc-queries-empty [simp]}: \text{count-wc-queries } \{\} \text{ sorter } = 0
\]

\[
\langle \text{proof} \rangle
\]

\[
\text{lemma } \text{count-wc-queries-aux}: \text{assumes } \land R. R \in Rs \Rightarrow sorter' R Rs \subseteq Rs' \text{ finite Rs'}
\]

8
shows \( \text{count-wc-queries } R s \text{ sorter } \leq \text{Max } \left( (\lambda R. \text{count-queries } R \left( \text{sorter'} R \right) )' R s \right) \)
\text{(proof)}

primrec eval-sorter :: '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 \( \text{card } \left( (\lambda R. \text{eval-sorter } R e)’ R s \right) \leq 2^\text{count-wc-queries } R s e \)
\text{(proof)}

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

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

2.2 Lower bounds on number of comparisons

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

theorem
fixes sorter :: 'a sorter and xs :: 'a list
assumes distinct: distinct xs
assumes sorter: \( \forall R. \text{linorder-on } (\text{set } xs) R \implies \text{is-sorting } R xs \left( \text{eval-sorter } R \text{ sorter} \right) \)
defines Rs ≡ \{ R. \text{linorder-on } (\text{set } xs) R \}
shows two-power-count-queries-ge: \( \text{fact } (\text{length } xs) \leq (2^\text{count-wc-queries } R s \text{ sorter}) \)
and count-queries-ge: \( \log 2 \left( \text{fact } (\text{length } xs) \right) \leq \text{real } \left( \text{count-wc-queries } R s \text{ sorter} \right) \)
\text{(proof)}

lemma ln-fact-big-o: \( (\lambda n. \ln (\text{fact } n)) - (\ln (2 * pi * n) / 2 + n * \ln n - n)) \in O(\lambda n. 1/n) \)
and asymp-equiv-ln-fact [asymp-equiv-intros]: \( (\lambda n. \ln (\text{fact } n)) \sim [\text{at-top}] (\lambda n. n * \ln n) \)
\text{(proof)}
include asymp-equiv-notation
\text{(proof)}
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 $\text{sorter} :: \text{nat} \Rightarrow \text{nat}$

assumes $\text{sorter} : \forall n. \text{linorder-on } \{..<n\} R \Rightarrow \text{is-sorting } R [0..<n] (\text{eval-sorter } R (\text{sorter } n))$

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⟩ end

**References**