Cost Analysis of QuickSort

Manuel Eberl

August 16, 2018

Abstract

We give a formal proof of the well-known results about the number of comparisons performed by two variants of QuickSort: first, the expected number of comparisons of randomised QuickSort (i.e. QuickSort with random pivot choice) is $2(n + 1)H_n - 4n$, which is asymptotically equivalent to $2n \ln n$; second, the number of comparisons performed by the classic non-randomised QuickSort has the same distribution in the average case as the randomised one.

Contents

1 Randomised QuickSort 2
   1.1 Deletion by index ........................................... 2
   1.2 Definition .................................................. 3
   1.3 Correctness proof ........................................... 4
   1.4 Cost analysis ............................................... 5
   1.5 Expected cost ............................................... 7
   1.6 Version for lists with repeated elements ............... 12

2 Average case analysis of deterministic QuickSort 17
   2.1 Definition of deterministic QuickSort .................. 17
   2.2 Analysis .................................................... 18
1 Randomised QuickSort

theory Randomised-Quick-Sort

imports
  HOL Probability
  Probability.Landau-Symbols
  Probability.Landau-More
  Comparison-Sort-Lower-Bound Linorder-Relations

begin

1.1 Deletion by index

The following function deletes the \( n \)-th element of a list.

fun delete-index :: nat ⇒ 'a list ⇒ 'a list
  where
  delete-index - [] = []
  | delete-index 0 (x # xs) = xs
  | delete-index (Suc n) (x # xs) = x # delete-index n xs

lemma delete-index-altdef: delete-index n xs = take n xs @ drop (Suc n) xs
  by (induction n xs rule: delete-index.induct) simp-all

lemma delete-index-ge-length: n ≥ length xs ⇒ delete-index n xs = xs
  by (simp add: delete-index-altdef)

lemma length-delete-index [simp]: n < length xs ⇒ length (delete-index n xs) = length xs - 1
  by (simp add: delete-index-altdef)

lemma delete-index-Cons: delete-index n (x # xs) = (if n = 0 then xs else x # delete-index (n - 1) xs)
  by (cases n) simp-all

lemma insert-set-delete-index: n < length xs ⇒ insert (xs ! n) (set (delete-index n xs)) = set xs
  by (induction n xs rule: delete-index.induct) auto

lemma add-mset-delete-index: i < length xs ⇒ add-mset (xs ! i) (mset (delete-index i xs)) = mset xs
  by (induction i xs rule: delete-index.induct) simp-all

lemma nth-delete-index: i < length xs ⇒ n < length xs ⇒
  delete-index n xs ! i = (if i < n then xs ! i else xs ! Suc i)
  by (auto simp: delete-index-altdef nth-append min-def)

lemma set-delete-index-distinct: assumes distinct xs n < length xs
  shows set (delete-index n xs) = set xs - {xs ! n}
  using assms by (induction n xs rule: delete-index.induct) fastforce+
lemma distinct-delete-index [simp, intro]:
assumes distinct xs
shows distinct (delete-index n xs)
proof (cases n < length xs)
case True
  with assms show ?thesis
    by (induction n xs rule: delete-index.induct) (auto simp: set-delete-index-distinct)
qed (simp-all add: delete-index-ge-length assms)

lemma mset-delete-index [simp]:
i < length xs =⇒ mset (delete-index i xs) = mset xs - {# xs!i #}
by (induction i xs rule: delete-index.induct) simp-all

1.2 Definition

The following is a functional randomised version of QuickSort that also
records the number of comparisons that were made. The randomisation is
in the selection of the pivot element: In each step, the next pivot is chosen
uniformly at random from all remaining list elements.

The function takes the ordering relation to use as a first argument in the
form of a set of pairs.

function rquicksort :: ('a × 'a) set ⇒ 'a list ⇒ ('a list × nat) pmf where
rquicksort R xs =
(if xs = [] then
  return-pmf ([], 0)
else
do {
  i ← pmf-of-set {..<length xs};
  let x = xs ! i;
  case partition (λy. (y,x) ∈ R) (delete-index i xs) of
  (ls, rs) ⇒ do {
    (ls, n1) ← rquicksort R ls;
    (rs, n2) ← rquicksort R rs;
    return-pmf (ls @ [x] @ rs, length xs - 1 + n1 + n2)
  }
} by auto
termination proof (relation Wellfounded.measure (length ∘ snd), goal-cases)
show wf (Wellfounded.measure (length ∘ snd)) by simp
qed (subst (asm) set-pmf-of-set; force intro!: le-less-trans[OF length-filter-le])+
declare rquicksort.simps [simp del]

lemma rquicksort-Nil [simp]: rquicksort R [] = return-pmf ([], 0)
by (simp add: rquicksort.simps)
1.3 Correctness proof

**Lemma:** \( \text{set-pmf-of-set-lessThan-length} \ [\text{simp}]: \)
\[
x \neq [] \implies \text{set-pmf} \ (\text{pmf-of-set} \ \{..<\text{length} \ x\}) = \{..<\text{length} \ x\}
\]

by (subst set-pmf-of-set) auto

We can now prove that any list that can be returned by QuickSort is sorted w.r.t. the given relation. (as long as that relation is reflexive, transitive, and total)

**Theorem:** \( \text{rquicksort-correct} \)
\[
\text{assumes} \ \text{trans} \ R \ \text{and} \ \text{total-on} \ (\text{set} \ x) \ \text{R} \ \text{and} \ \forall x \in \text{set} \ x, (x, x) \in R
\]
\[
\text{shows} \ \text{sorted-wrt} \ R \ yss \ \wedge \ \text{mset} \ yss = \text{mset} \ xss
\]

using \( \	ext{assms}(2-3) \)

**Proof:** (induction \( xss \) arbitrary: \( yss \ n \) rule: length-induct)

**Case:** \( 1 \ xss \)

have \( \text{IH}: \text{sorted-wrt} \ R \ zss \ \text{mset} \ yss \)

if \( (zss, n) \in \text{set-pmf} \ (\text{rquicksort} \ R \ yss) \) length \( yss \ < \text{length} \ xss \) set \( yss \subseteq \text{set} \ xss \) for \( zss \ yss \ n \)

using that \( 1.\text{IH} \) total-on-subset\((\text{OF} \ \text{prems}(1)) \ \text{that}(3)\) \( 1.\text{prems}(2) \) by blast+

show ?case

**Proof:** (cases \( xss = [] \))

**Case:** \( \text{False} \)

with \( 1.\text{prems} \) obtain \( ls \ rs \ n1 \ n2 \ i \) where *:
\[
i < \text{length} \ xss \ (ls, n1) \in \text{set-pmf} \ (\text{rquicksort} \ R \ \{y\leftarrow\text{delete-index} \ i \ xss \ (y, xss ! i) \in R\})
\]
\[
\text{ls} = \text{ls} @ [xss ! i] @ rs
\]

by (subst (asm) rquicksort.simps[of \- \ xss]) \( \text{auto simp: Let-def o-def} \)

**Note:** \( yss = \{yss = \text{ls} @ [xss ! i] @ rs\} \)

**Define:** \( ls' \) \text{where} \( ls' = [y\leftarrow\text{delete-index} \ i \ xss \ (y, xss ! i) \in R] \)

**Define:** \( rs' \) \text{where} \( rs' = [y\leftarrow\text{delete-index} \ i \ xss \ (y, xss ! i) \notin R] \)

from \( i < \text{length} \ xss \) have less: length \( ls' < \text{length} \ xss \) length \( rs' < \text{length} \ xss \)

unfolding \( \text{ls’-def} \ \text{rs’-def} \) by (intro le-less-trans[of \text{length-filter-le}; force]+)

have \( ls: (ls, n1) \in \text{set-pmf} \ (\text{rquicksort} \ R \ ls') \) and \( rs: (rs, n2) \in \text{set-pmf} \ (\text{rquicksort} \ R \ rs') \)

using * unfolding \( \text{ls’-def} \ \text{rs’-def} \) by blast+

**Have:** \( \text{subset: set} \ ls' \subseteq \text{set} \ xss \ \text{set} \ rs' \subseteq \text{set} \ xss \)

using insert-set-delete-index[of \ i \ xss] \( i < \text{length} \ xss \)

by (auto simp: \( \text{ls’-def} \ \text{rs’-def} \))

**Have:** \( \text{sorted: sorted-wrt} \ R \ ls \ \text{sorted-wrt} \ R \ rs \)

and \( \text{mset}: \ \text{mset} \ ls = \ \text{mset} \ ls' \ \text{mset} \ rs = \ \text{mset} \ rs' \)

by (rule \( \text{IH}[\text{of} \ \text{ls} \ n1 \ \text{ls'}] \ \text{IH}[\text{of} \ \text{rs} \ n2 \ \text{rs'}] \ \text{less} \ \text{ls} \ \text{rs} \ \text{subset} \))

**Have:** \( \text{ls-le:} (x, xss ! i) \in R \) if \( x \in \text{set} \ ls \) for \( x \)

**Proof:**

from that \( \text{have} \ x \in \# \ \text{mset} \ ls \) by simp

also note \( \text{mset}(1) \)

finally show ?thesis by (simp add: \( \text{ls’-def} \))

4
have \( rs\text{-ge} : (x, xs ! i) \notin R (xs ! i, x) \in R \) if \( x \in \text{set} \, rs \) for \( x \)

proof 
  
from that have \( x \in \# \, \text{mset} \, rs \) by simp
also note \( \text{mset}(2) \)
finally have \( x: x \in \text{set} \, rs' \) by simp
thus \( (x, xs ! i) \notin R \) by (simp-all add: \( \text{rs'-def} \))
from \( x \) and subset and \( (i < \text{length} \, xs) \) have \( x \in x \in t \) auto
with \( 1 \text{-prems} \) and \( (x, xs ! i) \notin R \) show \( (xs ! i, x) \in R \)

qed  

have \( \text{sorted-wrt} \, R \, ys \) unfolding \( \text{ys} \)
  
by (intro \text{sorted-wrt-append} \langle \text{trans} \, R \rangle \, \text{sorted-wrt-singleton} \, \text{sorted})
(auto intro: \text{rs-ge} \, \text{ls-le} \, \text{transD} [OF \langle \text{trans} \, R \rangle, \, of - \, xs ! i])
moreover have \( \text{mset} \, ys = \text{mset} \, xs \) unfolding \( \text{ys} \) using \( i < \text{length} \, xs \)
  
by (simp add: \text{mset} \, \text{ls'-def} \, \text{rs'-def} \, \text{add-mset-delete-index})
ultimately show \( \? \text{thesis} \) ..

qed (insert \( 1 \text{-prems}, \, \text{simp-all} \))

qed  

1.4 Cost analysis

The following distribution describes the number of comparisons made by
randomised QuickSort in terms of the list length. (This is only valid if all
list elements are distinct)

A succinct explanation of this cost analysis is given by Jacek Cichoń [1].

fun \( \text{rqs-cost} :: \text{nat} \Rightarrow \text{nat} \, \text{pmf} \) where
  \( \text{rqs-cost} \, 0 = \text{return-pmf} \, 0 \)
| \( \text{rqs-cost} \, (\text{Suc} \, n) = \)
  
d \{ i \leftarrow \text{pmf-of-set} \{..n\}; a \leftarrow \text{rqs-cost} \, i; b \leftarrow \text{rqs-cost} \, (n - i); \text{return-pmf} \, (n + a + b) \} 

lemma finite-set-pmf-rqs-cost [intro!]: finite \( \text{(set-pmf} \, (\text{rqs-cost} \, n)) \)
  
by (induction \( n \) rule: \( \text{rqs-cost.induct} \)) simp-all 

We connect the \( \text{rqs-cost} \) function to the \( \text{rquicksort} \) function by showing that
projecting out the number of comparisons from a run of \( \text{rquicksort} \) on a list
with distinct elements yields the same distribution as \( \text{rqs-cost} \) for the length
of that list.

theorem snd-rquicksort:
  
assumes \( \text{linorder-on} \, A \, R \) and \( \text{set} \, x \subseteq A \) and distinct \( x \)
shows \( \text{map-pmf} \, \text{snd} \, (\text{rquicksort} \, R \, x) = \text{rqs-cost} \, \text{(length} \, x) \)

using \( \text{assms}(2-) \)

proof (induction \( x \) rule: \( \text{length-induct} \))
  
case \( (1 \, x) \)
  
have \( \text{IH:} \, \text{map-pmf} \, \text{snd} \, (\text{rquicksort} \, R \, y) = \text{rqs-cost} \, \text{(length} \, y) \)
\[
\text{if } \text{length } ys < \text{length } xs \text{ mset } ys \subseteq \# \text{ mset } xs \text{ for } ys
\]

**proof**

\(\text{from } \text{set-mset-mono[OF that(2)] have } ys \subseteq set xs \text{ by simp} \)

also note \((set xs \subseteq A)\)

finally have \(set ys \subseteq A\).

moreover have \((\text{distinct } xs) \text{ and that(2) have } \text{distinct } ys\)

by \((\text{rule } \text{distinct-mset-mono})\)

ultimately show \(\text{?thesis using that and } 1.IH \text{ by blast}\)

qed

\begin{align*}
\text{define } n \text{ where } n = \text{length } zs \\
\text{define } \text{cnt where } \text{cnt} = (\lambda i. \text{length } [y \leftarrow \text{delete-index } i \text{ xs}. (y, x, s ! i) \in R]) \\
\text{have } \text{cnt-altdef}: \text{cnt } i = \text{linorder-rank } R (\text{set } xs) (x s ! i) \text{ if } i: i < n \text{ for } i
\end{align*}

**proof**

\begin{align*}
\text{have } \text{cnt } i = \text{length } [y \leftarrow \text{delete-index } i \text { xs}. (y, x, s ! i) \in R] \text{ by } \text{(simp add: cnt-def)} \\
\text{also have } \ldots = \text{card } (\text{set } [y \leftarrow \text{delete-index } i \text { xs}. (y, x, s ! i) \in R]) \\
\text{by } \text{(intro distinct-card [symmetric] distinct-filter distinct-delete-index 1.prems)} \\
\text{also have } \text{set } [y \leftarrow \text{delete-index } i \text { xs}. (y, x, s ! i) \in R] = \\
\{x \in \text{set } xs \setminus \{xs!i\}. (x, x, s ! i) \in R\}
\end{align*}

\[\text{using 1.prems and } i \text{ by } \text{(simp add: set-delete-index-distinct n-def)}\]

\[\text{also have } \text{card } \ldots = \text{linorder-rank } R (\text{set } xs) (x s ! i) \text{ by } \text{(simp add: linorder-rank-def)}\]

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

qed

from 1.prems have \(\text{bij-betw } ([!]) \text { xs } \{..<n\} \text{ (set } xs)\)

by \((\text{intro bij-betw-byWitness[where } f' = \text{index } xs])\) \text{(auto simp; n-def index-nth-id)}

moreover have \(\text{bij-betw } (\text{linorder-rank } R (\text{set } xs)) \{..<\text{card } (\text{set } xs)\}\)

\[\text{using assms(1) by } \text{(rule bij-betw-linorder-rank) (insert 1.prems, auto)}\]

ultimately have \(\text{bij-betw } (\text{linorder-rank } R (\text{set } xs) \circ (\lambda i. x s ! i)) \{..<n\} \{..<n\} \text{ (set } xs)\}

\[\text{by } \text{(rule bij-betw-trans)}\]

hence \(\text{bij: bij-betw } (\lambda i. \text{linorder-rank } R (\text{set } xs) (x s ! i)) \{..<n\} \{..<n\}\)

\[\text{using 1.prems by } \text{(simp add: n-def o-def distinct-card)}\]

show \(\text{?case}\)

**proof** \((\text{cases } xs = [])\)

case False

hence \(n > 0\) by \((\text{simp add: n-def})\)

hence \([\text{smp]}: n \neq 0\) by \((\text{intro notI})\) \text{auto}\)

\[\text{from False have } \text{map-pmf snd } (\text{rquicksort } R \text{ xs}) = \]
\[\text{map-pmf-of-set } \{..<\text{length } xs\} \Rightarrow \]
\[\text{(map-pmf } (\lambda i. \text{map-pmf } (\lambda z. \text{length } xs - 1 + \text{fst } z + \text{snd } z)) \]
\[\text{(pair-pmf } (\text{map-pmf } \text{snd } (\text{rquicksort } R [y \leftarrow \text{delete-index } i \text { xs}. (y, x, x, s ! i) \in R])))
\]
\[\text{by } (\text{subst rquicksort.simps}) \]
\[\text{by } (\text{simp add: map-bind-pmf bind-map-pmf Let-def case-prod-unfold} o-def\]
\[\text{pair-pmf-def})\]

6
also have ... = pmf-of-set {..<\text{length} \, \text{xs}} \gg \gg

\begin{align*}
(\lambda i. \, \text{map-pmf} \, (\lambda z. \, n - 1 + \text{fst} \, z + \text{snd} \, z)) \\
\quad \quad (\text{pair-pmf} \, (\text{rqs-cost} \, (\text{cnt} \, i)) \, (\text{rqs-cost} \, (n - 1 - \text{cnt} \, i))))
\end{align*}

proof (intro bind-pmf-cong refl, goal-cases)
\begin{itemize}
  \item case (1 \, i)
  \begin{itemize}
    \item with \{ \text{xs} \notin [\emptyset] \} have \, i: \, i < \text{length} \, \text{xs} \, \text{by} \, \text{auto}
    \begin{itemize}
      \item from \, i \, \text{have} \, \text{map-pmf} \, \text{snd} \, (\text{rquicksort} \, R \, [y \leftarrow \text{delete-index} \, i \, \text{xs} \, . \, (y, \, x s \, ! \, i) \notin R]) = \, \text{rqs-cost} \, (\text{length} \, [y \leftarrow \text{delete-index} \, i \, \text{xs} \, . \, (y, \, x s \, ! \, i) \notin R]) \, \text{by} \, (\text{intro \, IH})
      \begin{itemize}
        \item (auto \, intro!: \text{le-less-trans} [OF \text{length-filter-le}] \text{simp:} \text{mset-filter} \text{intro:} \text{subset-mset.order.trans} \text{multiset-filter-subset} \text{diff-subset-eq-self})
        \item also have \, \text{length} \, [y \leftarrow \text{delete-index} \, i \, \text{xs} \, . \, (y, \, x s \, ! \, i) \notin R] = n - 1 - \text{cnt} \, i \text{ unfolding} \, \text{n-def} \, \text{cnt-def using} \, \text{sum-length-filter-compl} [\text{of} \, \lambda y. \, (y, \, x s \, ! \, i) \in R \, \text{delete-index} \, i \, \text{xs}] \, \text{i} \, \text{by} \, \text{simp}
      \end{itemize}
      \end{itemize}
    \end{itemize}
  \end{itemize}
\item finally have \, \text{map-pmf} \, \text{snd} \, (\text{rquicksort} \, R \, [y \leftarrow \text{delete-index} \, i \, \text{xs} \, . \, (y, \, x s \, ! \, i) \notin R]) = \, \text{rqs-cost} \, (n - 1 - \text{cnt} \, i) .
\item moreover have \, \text{map-pmf} \, \text{snd} \, (\text{rquicksort} \, R \, [y \leftarrow \text{delete-index} \, i \, \text{xs} \, . \, (y, \, x s \, ! \, i) \in R]) = \, \text{rqs-cost} \, (\text{cnt} \, i) \, \text{unfolding} \, \text{cnt-def using} \, \text{i}
\item by \, (\text{intro \, IH})
\begin{itemize}
  \item (auto \, intro!: \text{le-less-trans} [OF \text{length-filter-le}] \text{simp:} \text{mset-filter} \text{intro:} \text{subset-mset.order.trans} \text{multiset-filter-subset} \text{diff-subset-eq-self})
\end{itemize}
\item ultimately show \, \text{?case by} \, (\text{simp only:} \, \text{n-def})
\item qed
\item also have ... = \text{map-pmf} \, \text{cnt} \, (\text{pmf-of-set} \, \{\cdot.<\text{length} \, \text{xs}\}) \gg \gg

\begin{align*}
(\lambda i. \, \text{map-pmf} \, (\lambda z. \, n - 1 + \text{fst} \, z + \text{snd} \, z)) \\
\quad \quad \text{(pair-pmf} \, (\text{rqs-cost} \, (\text{cnt} \, i)) \, (\text{rqs-cost} \, (n - 1 - \text{cnt} \, i))))
\end{align*}

\begin{itemize}
  \item (is = \text{bind-pmf} - \text{?f}) \, \text{by} \, (\text{simp add:} \, \text{bind-map-pmf} \, \text{n-def})
\end{itemize}
\item also have \, \text{map-pmf} \, \text{cnt} \, (\text{pmf-of-set} \, \{\cdot.<\text{n}\}) = \, \text{map-pmf} \, (\lambda i. \, \text{linorder-rank} \, R \, (\text{set} \, \text{xs}) \, (x s \, ! \, i)) \, (\text{pmf-of-set} \, \{\cdot.<\text{n}\}) \, \text{using} \, (n > 0) \, \text{by} \, (\text{intro \, map-pmf-cong \, refl, \, subst} \, (asm) \, \text{set-pmf-of-set}) \, (\text{auto \, simp:} \, \text{cnt-altdef})
\item also from (n > 0) \, \text{have} ... = \text{pmf-of-set} \, \{\cdot.<\text{n}\} \, \text{by} \, (\text{intro \, map-pmf-of-set-bij-betw bij \, auto})
\item also have \, \text{pmf-of-set} \, \{\cdot.<\text{n}\} \gg \gg \text{?f} = \text{rqs-cost} \, \text{n}
\item by \, (cases \, n) \, (\text{simp-all add:} \, \text{lessThan-Suc-atMost} \, \text{bind-map-pmf} \, \text{map-bind-pmf} \, \text{pair-pmf-def})
\item finally show \, \text{?thesis by} \, (\text{simp add:} \, \text{n-def})
\item qed \, \text{simp-all}
\item qed
\end{itemize}

1.5 Expected cost

It is relatively straightforward to see that the following recursive function (sometimes called the ‘QuickSort equation’) describes the expectation of \text{rqs-cost}, i.e. the expected number of comparisons of QuickSort when run on
a list with distinct elements.

fun rqs-cost-exp :: nat ⇒ real where
  rqs-cost-exp 0 = 0
| rqs-cost-exp (Suc n) = real n + (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n - i)) / real (Suc n)

lemmas rqs-cost-exp-0 = rqs-cost-exp.simps(1)
lemmas rqs-cost-exp-Suc [simp del] = rqs-cost-exp.simps(2)
lemma rqs-cost-exp-Suc-0 [simp]: rqs-cost-exp (Suc 0) = 0 by (simp add: rqs-cost-exp-Suc)

The following theorem shows that rqs-cost-exp is indeed the expectation of rqs-cost.

theorem expectation-rqs-cost: measure-pmf.expectation (rqs-cost n) real = rqs-cost-exp n
proof (induction n rule: rqs-cost.induct)
  case (2 n)
  note IH = 2.IH
  have measure-pmf.expectation (rqs-cost (Suc n)) real =
    (∑ a≤n. inverse (real (Suc n)) *
      measure-pmf.expectation (rqs-cost a) ≥≤ λa. rqs-cost (n - a) ≥≥
      (λb. return-pmf (n + aa + b))) real
    unfolding rqs-cost.simps by (subst pmf-expectation-bind-pmf-of-set) auto
  also have ... = (∑ i≤n. inverse (real (Suc n)) * (real n + rqs-cost-exp i + rqs-cost-exp (n - i)))
  proof (intro sum.cong refl, goal-cases)
    case (1 i)
    have rqs-cost i ≥≥ (λa. rqs-cost (n - i) ≥≥ (λb. return-pmf (n + a + b))) =
      map-pmf (λ(a,b). n + a + b) (pair-pmf (rqs-cost i) (rqs-cost (n - i)))
    by (simp add: pair-pmf-def map-bind-pmf)
    also have measure-pmf.expectation ... real =
      measure-pmf.expectation (pair-pmf (rqs-cost i) (rqs-cost (n - i)))
      (λz. real n + (real (fst z) + real (snd z)))
    by (subst integral-map-pmf) (simp add: case_prod_unfold add_ac)
    also have ... = real n + measure-pmf.expectation (pair-pmf (rqs-cost i)
      (rqs-cost (n - i)))
    by (subst Bochner-Integration.integral-add) (auto intro: integrable-measure-pmf-finite)
    also have ?A = measure-pmf.expectation (map-pmf fst (pair-pmf (rqs-cost i))
      (rqs-cost (n - i))) real +
    measure-pmf.expectation (map-pmf snd (pair-pmf (rqs-cost i))
      (rqs-cost (n - i))) real
    unfolding integral-map-pmf
    by (subst Bochner-Integration.integral-add) (auto intro: integrable-measure-pmf-finite)
    also have ... = measure-pmf.expectation (rqs-cost i) real +
    measure-pmf.expectation (rqs-cost (n - i)) real
    unfolding map-fst-pair-pmf map-snd-pair-pmf ..
    also from 1 have ... = rqs-cost-exp i + rqs-cost-exp (n - i) by (simp-all add: IH)
finally show ?case by simp

qed
also have \( \ldots = (\sum_{i \leq n. \text{inverse } (\text{real } (\text{Suc } n))} \ast \text{real } n) + \)
\( (\sum_{i \leq n. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i)}) \) / \text{real } (\text{Suc } n)
by (simp add: sum-distrib field-simps sum-distrib-left sum-distrib-right
sum-divide-distrib [symmetric] del: of-nat-Suc)
also have \( (\sum_{i \leq n. \text{rqs-cost-exp } i \ast \text{rqs-cost-exp } (n - i)}) = \text{rqs-cost-exp } (\text{Suc } n) \)
by (simp add: rqs-cost-exp-Suc)
finally show ?thesis.

qed simp-all

We will now obtain a closed-form solution for \text{rqs-cost-exp}. First of all, we
can reindex the right-most sum in the recursion step and obtain:

\textbf{lemma rqs-cost-exp-Suc':}
\text{rqs-cost-exp } (\text{Suc } n) = \text{real } n + 2 / \text{real } (\text{Suc } n) \ast (\sum_{i \leq n. \text{rqs-cost-exp } i})
\textbf{proof –}

have \text{rqs-cost-exp } (\text{Suc } n) = \text{real } n + (\sum_{i \leq n. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i)}) / \text{real } (\text{Suc } n)
by (rule rqs-cost-exp-Suc)
also have \( (\sum_{i \leq n. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i)}) = (\sum_{i \leq n. \text{rqs-cost-exp } i}) + (\sum_{i \leq n. \text{rqs-cost-exp } (n - i)}) \)
by (simp add: sum-distrib)
also have \( (\sum_{i \leq n. \text{rqs-cost-exp } (n - i)}) = (\sum_{i \leq n. \text{rqs-cost-exp } i}) \)
by (intro sum.reindex-bij-witness[of - \lambda i. n - i \lambda i. n - i] auto)
also have \( \ldots + \) \( \ldots = 2 \ast \ldots \) by simp
also have \( \ldots / \text{real } (\text{Suc } n) = 2 / \text{real } (\text{Suc } n) \ast (\sum_{i \leq n. \text{rqs-cost-exp } i}) \) by simp
finally show ?thesis.

qed

Next, we can apply some standard techniques to transform this equation
into a simple linear recurrence, which we can then solve easily in terms of
harmonic numbers:

\textbf{theorem rqs-cost-exp-eq [code]:}
\text{rqs-cost-exp } n = 2 \ast \text{real } (n + 1) \ast \text{harm } n - 4
\ast \text{real } n
\textbf{proof –}

define \( F \) where \( F = (\lambda n. \text{rqs-cost-exp } n / (\text{real } n + 1)) \)
have [simp]: \( F 0 = 0 F (\text{Suc } 0) = 0 \) by (simp-all add: F-def)
have \( F\text{-Suc}: F (\text{Suc } m) = F m + \text{real } (2m) / (\text{real } ((m + 1) * (m + 2))) \) if \( m > 0 \) for \( m \)
proof (cases \( m \))

case (Suc \( n \))
have \( A: \text{rqs-cost-exp } (\text{Suc } (\text{Suc } n)) \ast \text{real } (\text{Suc } (\text{Suc } n)) = \)
\( \text{real } ((n + 1) * (n + 2)) + 2 \ast (\sum_{i \leq n. \text{rqs-cost-exp } i}) + 2 \ast \text{rqs-cost-exp } (\text{Suc } n) \)
by (subst rqs-cost-exp-Suc') (simp-all add: field-simps)
have B: \text{rqs-cost-exp} (Suc n) * real (Suc n) = real (n*(n+1)) + 2 * (\sum i\leq n. \text{rqs-cost-exp} i)

by (subst \text{rqs-cost-exp-Suc'}) (simp-all add: field-simps)

have \text{rqs-cost-exp} (Suc (Suc n)) * real (Suc (Suc n)) = \text{rqs-cost-exp} (Suc n) * real (Suc n) =
real ((n+1)*(n+2)) − real (n*(n+1)) + 2 * \text{rqs-cost-exp} (Suc n)

by (subst A, subst B) simp-all

also have real ((n+1)*(n+2)) − real (n*(n+1)) = real (2*(n+1)) by simp

finally have \text{rqs-cost-exp} (Suc (Suc n)) * real (n+2) = \text{rqs-cost-exp} (Suc n) * real (n+3) + real (2*(n+1))

by (simp add: algebra-simps)

hence \text{rqs-cost-exp} (Suc (Suc n)) / real (n+3) = \text{rqs-cost-exp} (Suc n) / real (n+2) + real (2*(n+1)) / (real (n+2)*real (n+3))

by (simp add: divide-simps del: of-nat-Suc of-nat-add)

thus \text{thesis} by (simp add: F-def algebra-simps Suc)

qed simp-all

have \text{F-eq}: F n = 2 * (\sum k=1..n. real (k − 1) / real (k * (k + 1))) for n

proof (cases n \geq 1)

case True

thus \text{thesis} by (induction n rule: dec-induct) (simp-all add: F-Suc algebra-simps)

qed (simp-all add: not-le)

have F n = 2 * (\sum k=1..n. real (k − 1) / real (k * (k + 1))) (is - = 2 * ?S)

by (fact \text{F-eq})

also have \text{cases} ?S = (\sum k=1..n. 2 / real (Suc k) − 1 / real k)

by (intro sum.cong) (simp-all add: field-simps of-nat-diff)

also have \ldots = 2 * (\sum k=1..n. inverse (real (Suc k))) − harm n

by (subst sum-subtractf) (simp add: harm-def sum.distrib sum.distrib-left divide-simps)

also have (\sum k=1..n. inverse (real (Suc k))) = (\sum k=Suc 1..Suc n. inverse (real k))

by (intro sum.reindex-bij-witness[of - \lambda x. x = 1 Suc]) auto

also have \ldots = harm (Suc n) − 1 unfolding harm-def by (subst (2) sum-head-Suc)

simp-all

finally have F n = 2 * harm n + 4 * (1 / (n + 1) − 1) by (simp add: harm-Suc field-simps)

also have \ldots * real (n + 1) = 2 * real (n + 1) * harm n − 4 * real n

by (simp add: field-simps)

also have F n * real (n + 1) = \text{rqs-cost-exp} n by (simp add: F-def add-ac)

finally show \text{thesis}.

qed

lemma asymp-eqiv-harm [asymp-equiv-intros]: harm \sim [at-top] (\lambda n. ln (real n))

proof

have (\lambda n. harm n − ln (real n)) \in O(\lambda-. 1) using euler-mascheroni-LIMSEQ

by (intro bigo-tendsto[where c = euler-mascheroni]) simp-all

also have (\lambda-. 1) \in o(\lambda n. ln (real n)) by auto

10
finally have \((\lambda n. \ln (\text{real } n) + (\text{harm } n - \ln (\text{real } n))) \sim_{\text{at-top}} (\lambda n. \ln (\text{real } n))\)
   by (subst asymp-equiv-add-right) simp-all
thus \(?thesis by simp
qed

corollary rqs-cost-exp-asymp-equiv: rqs-cost-exp \sim_{\text{at-top}} (\lambda n. 2 \times n \times \ln n)
proof –
  have rqs-cost-exp = (\lambda n. 2 \times \text{real } (n + 1) \times \text{harm } n - 4 \times \text{real } n) using rqs-cost-exp-eq ..
  also have \ldots = (\lambda n. 2 \times \text{real } n \times \text{harm } n + (2 \times \text{harm } n - 4 \times \text{real } n))
     by (simp add: algebra-simps)
finally have rqs-cost-exp \sim_{\text{at-top}} \ldots by simp
also have \ldots \sim_{\text{at-top}} (\lambda n. 2 \times \text{real } n \times \text{harm } n)
proof (subst asymp-equiv-add-right)
  have \((\lambda x. 1 \times \text{harm } x) \in o(\lambda x. \text{real } x \times \text{harm } x)\)
     by (intro landau-o_small-big-mult smalllo-real-nat-transfer) simp-all
moreover have \text{harm} \in \omega(\lambda x. 1 :: \text{real})
     by (intro smallomegaI_filterlim_at_top_norm) (auto simp: harm-at-top)
  hence \((\lambda x. \text{real } x \times 1) \in o(\lambda x. \text{real } x \times \text{harm } x)\)
     by (intro landau-o_big-small-mult) (simp-all add: smallomega_iff_smallo)
ultimately show \((\lambda n. 2 \times \text{harm } n - 4 \times \text{real } n) \in o(\lambda n. 2 \times \text{real } n \times \text{harm } n)\)
     by (intro sum-in-smallo) simp-all
qed simp-all
also have \ldots \sim_{\text{at-top}} (\lambda n. 2 \times \text{real } n \times \ln (\text{real } n)) by (intro asymp-equiv-intros)
finally show \(?thesis
qed

lemma harm-mono: \(m \leq n \Longrightarrow \text{harm } m \leq (\text{harm } n :: \text{real})\)
unfolding harm-def by (intro sum-mono2) auto

lemma harm-Suc-0 [simp]: \(\text{harm } (\text{Suc } 0) = 1\)
by (simp add: harm-def)

lemma harm-ge-1: \(n > 0 \Longrightarrow \text{harm } n \geq (1 :: \text{real})\)
using harm-mono[of 1 n] by simp

lemma mono-rqs-cost-exp: mono rqs-cost-exp
proof (rule incseq-SucI)
  fix \(n\) show rqs-cost-exp \(n\) \(\leq\) rqs-cost-exp (Suc \(n\))
proof (cases \(n = 0\))
  case False
  have \(0 < (1 \times 2 \times (\text{real } n + 1) - 2 \times \text{real } n) / (\text{real } n + 1)\) by simp
  also have \ldots \(\leq\) \((\text{harm } n \times 2 \times (\text{real } n + 1) - 2 \times \text{real } n) / (\text{real } n + 1)\)
using False
     by (intro divide-right-mono diff-right-mono mult-right-mono) (auto simp: harm-ge-1)
  also have \ldots = rqs-cost-exp (Suc \(n\)) - rqs-cost-exp \(n\)
by (simp add: rqs-cost-exp-eq harm-Suc field-simps)
finally show thesis by simp
qed auto

lemma rqs-cost-exp-leI: m ≤ n ⟹ rqs-cost-exp m ≤ rqs-cost-exp n
using mono-rqs-cost-exp by (simp add: mono-def)

1.6 Version for lists with repeated elements

definition threeway-partition where
threeway-partition x R xs =
(filter (λy. (y,x) ∈ R ∧ (x,y) ∉ R) xs,
filter (λy. (x,y) ∈ R ∧ (y,x) ∈ R) xs,
filter (λy. (x,y) ∈ R ∧ (y,x) ∉ R) xs)

The following version of randomised Quicksort uses a three-way partitioning function in order to also achieve expected logarithmic running time on lists with repeated elements.

function rquicksort' :: ('a × 'a) set ⇒ 'a list ⇒ ('a list × nat) pmf where
rquicksort' R xs =
(if xs = [] then
  return-pmf ([], 0)
else
do {
i ← pmf-of-set {..<length xs};
  let x = xs ! i;
  case threeway-partition x R (delete-index i xs) of
    (ls, es, rs) ⇒ do {
    (ls, n1) ← rquicksort' R ls;
    (rs, n2) ← rquicksort' R rs;
    return-pmf (ls @ x # es @ rs, length xs - 1 + n1 + n2)
  }
}
by auto

termination proof (relation Wellfounded.measure (length o snd), goal-cases)
show wf (Wellfounded.measure (length o snd)) by simp
qed (subst (asm) set-pmf-of-set;
  force intro!: le-less-trans[OF length-filter-le] simp: threeway-partition-def)+
declare rquicksort'.simp's [simp del]

lemma rquicksort'-Nil [simp]: rquicksort' R [] = return-pmf ([], 0)
by (simp add: rquicksort'.simp's)

context
begin

qualified definition lesss :: ('a × 'a) set ⇒ 'a ⇒ 'a list ⇒ 'a list where
\[ \text{lesss } R \; x \; x s = \text{filter } (\lambda y. (y, x) \in R \land (x, y) \notin R) \; x s \]

**qualified definition** \n\[ \text{greaters } :: (\; 'a \times 'a \; \text{set} \Rightarrow 'a \Rightarrow 'a \; \text{list} \) \text{ where} \]
\[ \text{greaters } R \; x \; x s = \text{filter } (\lambda y. (x, y) \in R \land (y, x) \notin R) \; x s \]

**qualified lemma** \n\[ \text{lesss-Cons: } \]
\[ \text{lesss } R \; x \; (y \# y s) = \]
\[ (\text{if } (y, x) \in R \land (x, y) \notin R \text{ then } y \# \text{less } R \; x \; y s \text{ else } \text{less } R \; x \; y s) \]
\[ \text{by } (\text{simp add: lesss-def}) \]

**qualified lemma** \n\[ \text{length-lesss-le [intro]: } \text{length } (\text{less } R \; x \; x s) \leq \text{length } x s \]
\[ \text{by } (\text{simp add: lesss-def}) \]

**qualified lemma** \n\[ \text{length-lesss-less [intro]: } \]
\[ \text{assumes } x \in \text{set } x s \]
\[ \text{shows } \text{length } (\text{less } R \; x \; x s) < \text{length } x s \]
\[ \text{using } \text{assms by } (\text{induction } x s) \; (\text{auto simp: lesss-Cons intro: le-less-trans}) \]

**qualified lemma** \n\[ \text{greaters-Cons: } \]
\[ \text{greaters } R \; x \; (y \# y s) = \]
\[ (\text{if } (x, y) \in R \land (y, x) \notin R \text{ then } y \# \text{greaters } R \; x \; y s \text{ else } \text{greaters } R \; x \; y s) \]
\[ \text{by } (\text{simp add: greaters-def}) \]

**qualified lemma** \n\[ \text{length-greaters-le [intro]: } \text{length } (\text{greaters } R \; x \; x s) \leq \text{length } x s \]
\[ \text{by } (\text{simp add: greaters-def}) \]

**qualified lemma** \n\[ \text{length-greaters-less [intro]: } \]
\[ \text{assumes } x \in \text{set } x s \]
\[ \text{shows } \text{length } (\text{greaters } R \; x \; x s) < \text{length } x s \]
\[ \text{using } \text{assms by } (\text{induction } x s) \; (\text{auto simp: greaters-Cons intro: le-less-trans}) \]

The following function counts the comparisons made by the modified randomised Quicksort.

**function** \n\[ \text{rqs’-cost } :: (\; 'a \times 'a \; \text{set} \Rightarrow 'a \Rightarrow 'a \; \text{list} \Rightarrow \text{nat } \text{pmf} \) \text{ where} \]
\[ \text{rqs’-cost } R \; x s = \]
\[ (\text{if } x s = [] \text{ then } \text{return-pmf } 0 \text{ else } \text{do } \{
\text{\hspace{1cm}}i \leftarrow \text{pmf-of-set} \{..<\text{length } x s\};
\text{\hspace{1cm}}\text{let } x = x s ![i];
\text{\hspace{1cm}}\text{map-pmf } (\lambda (n1, n2). \text{length } x s - 1 + n1 + n2)
\text{\hspace{1cm}}(\text{pair-pmf } (\text{rqs’-cost } R \; (\text{less } R \; x \; x s)) \; (\text{rqs’-cost } R \; (\text{greaters } R \; x \; x s)))
\text{\hspace{1cm}}\}) \text{ by auto}\]
\[ \text{termination by } (\text{relation Wellfounded.measure } (\text{length } \circ \text{snd})) \text{ auto}\]

**declare** \n\[ \text{rqs’-cost.simps [simp def]} \]

13
lemma \textit{rqs}'\text{-cost-nonempty}:
\[
xs \neq [] \implies \text{rqs}'\text{-cost } R \text{ } xs = \\
\begin{array}{l}
    \text{do } \{ \\
    \quad i \leftarrow \text{pmf-of-set } \{..<\text{length } xs\}; \\
    \quad \text{let } x = xs ! i; \\
    \quad n1 \leftarrow \text{rqs}'\text{-cost } R (\text{lesss } R x xs); \\
    \quad n2 \leftarrow \text{rqs}'\text{-cost } R (\text{greaters } R x xs); \\
    \quad \text{return-pmf } (\text{length } xs - 1 + n1 + n2)
    \} \\
\end{array}
\]
\by (\text{subst rqs}'\text{-cost.simps}) (\text{auto simp: pair-pmf-def Let-def map-bind-pmf})

lemma \textit{finite-set-pmf-rqs}'\text{-cost} [simp, intro]:
\[
\text{finite } (\text{set-pmf } (\text{rqs}'\text{-cost } R \text{ } xs))
\]
\by (\text{induction } R \text{ } xs \text{ rule: rqs}'\text{-cost.induct}) (\text{auto simp: rqs}'\text{-cost.simps Let-def})

lemma \textit{expectation-pair-pmf-fst} [simp]:
\[
\text{fixes } f :: 'a \Rightarrow 'b::{banach, second-countable-topology} \\
\text{shows } \text{measure-pmf.} \text{expectation } (\text{pair-pmf } p \text{ } q) (\lambda x. f (\text{fst } x)) = \text{measure-pmf.} \text{expectation } p \text{ } f
\]
\proof\begin{array}{l}
    \text{have } \text{measure-pmf.} \text{expectation } (\text{pair-pmf } p \text{ } q) (\lambda x. f (\text{fst } x)) = \\
    \quad \text{measure-pmf.} \text{expectation } (\text{map-pmf } \text{fst } (\text{pair-pmf } p \text{ } q)) \text{ } f \by \text{simp}
    \end{array}
\end{array}
\by (\text{simp add: map-fst-pair-pmf})
\finally \text{show } \text{?thesis .}
\qed

lemma \textit{expectation-pair-pmf-snd} [simp]:
\[
\text{fixes } f :: 'a \Rightarrow 'b::{banach, second-countable-topology} \\
\text{shows } \text{measure-pmf.} \text{expectation } (\text{pair-pmf } p \text{ } q) (\lambda x. f (\text{snd } x)) = \text{measure-pmf.} \text{expectation } q \text{ } f
\]
\proof\begin{array}{l}
    \text{have } \text{measure-pmf.} \text{expectation } (\text{pair-pmf } p \text{ } q) (\lambda x. f (\text{snd } x)) = \\
    \quad \text{measure-pmf.} \text{expectation } (\text{map-pmf } \text{snd } (\text{pair-pmf } p \text{ } q)) \text{ } f \by \text{simp}
    \end{array}
\end{array}
\by (\text{simp add: map-snd-pair-pmf})
\finally \text{show } \text{?thesis .}
\qed

qualified lemma \textit{length-lesss-le-sorted}:
\text{assumes } \text{sorted-wrt } R \text{ } xs \text{ } i < \text{length } xs
\text{shows } \text{length } (\text{lesss } R \text{ } (xs ! i) \text{ } xs) \leq i
\text{using } \text{assms} \by (\text{induction arbitrary: } i \text{ rule: sorted-wrt.induct})
\end{array}\end{array}\end{array}+ (\text{force simp: lesss-def nth-Cons le-Suc-eq split: nat.splits})+

qualified lemma \textit{length-greaters-le-sorted}:
\text{assumes } \text{sorted-wrt } R \text{ } xs \text{ } i < \text{length } xs
\text{shows } \text{length } (\text{greaters } R \text{ } (xs ! i) \text{ } xs) \leq \text{length } xs - i - 1
using assms
by (induction arbitrary: i rule: sorted-wrt.induct)
  (force simp: greaters-def nth-Cons le-Suc-eq split: nat.splits)+

qualified lemma length-lesss-le':
assumes i < length xs linorder-on A R set xs ⊆ A
shows length (lesss R (insort-wrt R xs ! i) xs) ≤ i
proof –
define x where x = insort-wrt R xs ! i
define less where less = (λx y. (x,y) ∈ R ∧ (y,x) /∈ R)
have length (lesss R x xs) = size \{ # y ∈# mset xs. less y x #\}
  by (simp add: lesss-def size-mset [symmetric] less-def mset-filter del: size-mset)
also have mset xs = mset (insort-wrt R xs) by simp
also have size \{ #y ∈# mset (insort-wrt R xs). less y x #\} =
  length (lesss R x (insort-wrt R xs))
  by (simp only: mset-filter [symmetric] size-mset lesss-def less-def)
also have ... ≤ i unfolding x-def by (rule length-lesss-le-sorted) (use assms in auto)
finally show ?thesis unfolding x-def .
qed

qualified lemma length-greaters-le':
assumes i < length xs linorder-on A R set xs ⊆ A
shows length (greaters R (insort-wrt R xs ! i) xs) ≤ length xs − i − 1
proof –
define x where x = insort-wrt R xs ! i
define less where less = (λx y. (x,y) ∈ R ∧ (y,x) /∈ R)
have length (greaters R x xs) = size \{ # y ∈# mset xs. less x y #\}
  by (simp add: greaters-def size-mset [symmetric] less-def mset-filter del: size-mset)
also have mset xs = mset (insort-wrt R xs) by simp
also have size \{ #y ∈# mset (insort-wrt R xs). less x y #\} =
  length (greaters R x (insort-wrt R xs))
  by (simp only: mset-filter [symmetric] size-mset greaters-def less-def)
also have ... ≤ length (insort-wrt R xs) − i − 1 unfolding x-def
  by (rule length-greaters-le-sorted) (use assms in auto)
finally show ?thesis unfolding x-def by simp
qed

We can show quite easily that the expected number of comparisons in this
modified QuickSort is bounded above by the expected number of comparisons on a list of the same length with no repeated elements.

theorem rqs'-cost-expectation-le:
assumes linorder-on A R set xs ⊆ A
shows measure-pmf.expectation (rqs'-cost R xs) real ≤ rqs-cost-exp (length xs)
using assms
proof (induction R xs rule: rqs'-cost.induct)
case (1 R xs)
show ?case
proof (cases xs = [])
case False

define n where n = length xs - 1

have length-eq: length xs = Suc n using False by (simp add: n-def)

define E where E = (λxs. measure-pmf.expectation (rqs'-cost R xs) real)

define f where f = (λx. rqs-cost-exp (length (lesss R x xs)) + rqs-cost-exp (length (greaters R x xs)))

have rqs'-cost R xs =
  do |
    i ← pmf-of-set {..<length xs};
    map-pmf (λ(n1, y). length xs = Suc 0 + n1 + y) (pair-pmf (rqs'-cost R (lesss R (xs ! i) xs)) (rqs'-cost R (greaters R (xs ! i) xs)))
  |
  using False by (subst rqs'-cost.simps) (simp-all add: Let-def)
also have measure-pmf.expectation ... real = real n + (∑ k<length xs. E (lesss R (xs ! k) xs) + E (greaters R (xs ! k) xs)) / real (length xs)
using False
by (subst pmf-expectation-bind-pmf-of-set)
also have ... ≤ real n + (∑ k<length xs. f (xs ! k)) / real (length xs)
unfolding E-def f-def using False 1.prems
by (intro add-mono order.refl divide-right-mono sum-mono 1.IH[OF - - refl]
False)

(auto simp: less-def greaters-def)
also have (∑ k<length xs. f (xs ! k)) = (∑ x∈#mset xs. f x)
by (simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth)
  (simp-all add: atLeast0LessThan)
also have mset xs = mset (insort-wrt R xs)
by simp
also have (∑ x∈#. f x) = (∑ i<length xs. f (insort-wrt R xs ! i))
by (simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth)
  (simp-all add: atLeast0LessThan)
also have ... ≤ (∑ i<length xs. rqs-cost-exp i + rqs-cost-exp (length xs - i - 1))
unfolding f-def

proof (intro sum-mono add-mono rqs-cost-exp-leI)
fix i assume i: i ∈ {..<length xs}
  show length (lesss R (insort-wrt R xs ! i) xs) ≤ i
    using 1.prems by (intro length-lesss-le[where A = A]) auto
  show length (greaters R (insort-wrt R xs ! i) xs) ≤ length xs - i - 1
    using 1.prems by (intro length-greaters-le[where A = A]) auto
qed
also have ... = (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n - i))
by (intro sum.cong) (auto simp: length-eq)
also have real n + ... / real (length xs) = rqs-cost-exp (length xs)
by (simp add: length-eq rqs-cost-exp-simps(2))
2 Average case analysis of deterministic QuickSort

theory Quick-Sort-Average-Case
  imports Randomised-Quick-Sort
begin

2.1 Definition of deterministic QuickSort

This is the functional description of the standard variant of deterministic
QuickSort that always chooses the first list element as the pivot as given by
Hoare in 1962 [2]. For a list that is already sorted, this leads to \( n(n-1) \)
comparisons, but as is well known, the average case is not that bad.

fun quicksort :: ('a × 'a) set ⇒ 'a list ⇒ 'a list where
  quicksort - [] = []
  | quicksort R (x # xs) =
      quicksort R (filter (λy. (y,x) ∈ R) xs) @ [x] @ quicksort R (filter (λy. (y,x) /∈ R) xs)

We can easily show that this QuickSort is correct:

theorem mset-quicksort [simp]: mset (quicksort R xs) = mset xs
by (induction R xs rule: quicksort.induct) simp-all

corollary set-quicksort [simp]: set (quicksort R xs) = set xs
by (induction R xs rule: quicksort.induct) auto

theorem sorted-wrt-quicksort:
  assumes trans R and total-on (set xs) R and \( \forall x. x ∈ set xs \Rightarrow (x, x) ∈ R \)
  shows sorted-wrt R (quicksort R xs)
using assms
proof (induction R xs rule: quicksort.induct)
  case (2 R x xs)
  have total: (a, b) ∈ R if (b, a) /∈ R a ∈ set (x#xs) b ∈ set (x#xs) for a b
    using 2.prems that unfolding total-on-def by (cases a = b) auto
  have *: sorted-wrt R (quicksort R (filter (λy. (y,x) ∈ R) xs))
    sorted-wrt R (quicksort R (filter (λy. (y,x) /∈ R) xs))
  by ((rule 2 total-on-subset[OF (total-on (set (x#xs)) R)]) | force)+
  show ?case
    by (auto intro!: sorted-wrt-append sorted-wrt.intros (trans R) *)
***intro: transD(OF ⟨trans R⟩ dest!: total simp: total-on-def)***

**Qed** auto

**Corollary** sorted-wrt-quicksort′:

**Assumes** linorder-on A R and set xs ⊆ A

**Shows** sorted-wrt R (quicksort R xs)

**By** (rule sorted-wrt-quicksort)

(insert assms, auto simp: linorder-on-def refl-on-def dest: total-on-subset)

We now define another version of QuickSort that is identical to the previous one but also counts the number of comparisons that were made.

**Fun** quicksort′ :: (′a × ′a) set ⇒ ′a list ⇒ ′a list × nat where

quicksort′ - [] = ([], 0)

| quicksort′ R (x # xs) = (let (ls, rs) = partition (λy. (y, x) ∈ R) xs;

(ls′, n1) = quicksort′ R ls;

(rs′, n2) = quicksort′ R rs

in

(ls′ @ [x] @ rs′, length xs + n1 + n2))

For convenience, we also define a function that computes only the number of comparisons that were made and not the result list.

**Fun** qs-cost :: (′a × ′a) set ⇒ ′a list ⇒ nat where

qs-cost - [] = 0

| qs-cost R (x # xs) = length xs + qs-cost R (filter (λy. (y, x) ∈ R) xs) + qs-cost R (filter (λy. (y, x) ∉ R) xs)

It is obvious that the original QuickSort and the cost function are the projections of the cost-counting QuickSort.

**Lemma** fst-quicksort′ [simp]: fst (quicksort′ R xs) = quicksort R xs

**By** (induction R xs rule: quicksort.induct) (simp-all add: case-prod-unfold Let-def o-def)

**Lemma** snd-quicksort′ [simp]: snd (quicksort′ R xs) = qs-cost R xs

**By** (induction R xs rule: quicksort.induct) (simp-all add: case-prod-unfold Let-def o-def)

### 2.2 Analysis

We will reduce the average-case analysis to showing that it is essentially equivalent to the randomised QuickSort we analysed earlier. Similar, but more direct analyses are given by Hoare [2] and Sedgewick [3].

The proof is relatively straightforward – but still a bit messy. We show that the cost distribution of QuickSort run on a random permutation of a set of size n is exactly the same as that of randomised QuickSort being run on any fixed list of size n (which we analysed before):
**Theorem** `qs-cost-average-conv-rqs-cost`:

**Assumes**
- `finite A` and `linorder-on B R` and `A ⊆ B`

**Shows**
- `map-pmf (qs-cost R) (pmf-of-set (permutations-of-set A)) = rqs-cost (card A)`

**Using** `assms(1,3)`

**Proof** *(induction A rule: finite-psubset-induct)*

**Case** `(psubset A)`

**Show** `?case`

**Proof**
- **Cases** `A = {}`
  - **Case** `True`
    - Thus `?thesis` by `(simp add: pmf-of-set-singleton)`
  - **Next**
    - **Note** `A = (finite A; A ≠ {})`
    - **Define** `n` where `n = card A − 1`
    - From `A` have `pmf-of-set (permutations-of-set A) =`
      - `do { x ← pmf-of-set A; xs ← pmf-of-set (permutations-of-set (A − {x})); return-pmf (x ≠ xs) }
      - by (rule random-permutation-of-set)`
    - Also have `map-pmf (qs-cost R) ... =`
      - `do {
          x ← pmf-of-set A;
          xs ← pmf-of-set (permutations-of-set (A − {x}));
          return-pmf (x = xs + qs-cost R [y ← xs. (y, x) ∈ R] + qs-cost R)
        }
      - by `(simp add: map-bind-pmf)`
    - Also have `... = map-pmf (λm. n + m) {
      - `do {
          x ← pmf-of-set A;
          xs ← pmf-of-set (permutations-of-set (A − {x}));
          return-pmf (qs-cost R [y ← xs. (y, x) ∈ R] + qs-cost R [y ← xs. (y, x) ∉ R])
        } (is = map-pmf - ?X) using A unfolding n-def map-bind-pmf
      - by (intro bind-pmf-cong map-pmf-cong refl) (auto simp: length-finite-permutations-of-set)`
    - Also have `?X = do {
      - `x ← pmf-of-set A;
        ls, rs ← map-pmf (partition (λy. (y, x) ∈ R))
        (pmf-of-set (permutations-of-set (A − {x}))); return-pmf (qs-cost R ls + qs-cost R rs)
      - ) by (simp add: bind-map-pmf o-def)`
    - Also have `... = do {
      - `x ← pmf-of-set A;
        (n1, n2) ← pair-pmf
        (rqs-cost (linorder-rank R A x)) (rqs-cost (n − linorder-rank R A x));
        return-pmf (n1 + n2)
      - )
    - Proof (intro bind-pmf-cong refl, goal-cases)
      - **Case** `(1 x)`
        - Have `map-pmf (partition (λy. (y, x) ∈ R)) (pmf-of-set (permutations-of-set (A − {x})))`
\[ \Rightarrow (\lambda(ls, rs). \text{return-pmf} (qs-cost R ls + qs-cost R rs)) = \\
\text{map-pmf} (\lambda(n1, n2). n1 + n2) (\text{pair-pmf} \\
(\text{map-pmf} (qs-cost R) (\text{pmf-of-set} (\text{permutations-of-set} \{xa \in A - \{x\}. \\
(xa, x) \in R\})))) \\
(\text{map-pmf} (qs-cost R) (\text{pmf-of-set} (\text{permutations-of-set} \{xa \in A - \{x\}. \\
(xa, x) \notin R\})))) \\
(\text{is - = map-pmf - (pair-pmf ?X ?Y)}) \\
\text{by (subst partition-random-permutations)} \\
(\text{simp-all add: map-pmf-def case-prod-unfold bind-return-pmf bind-assoc-pmf}) \\
\text{pair-pmf-def A}) \\
\text{also} \{ \\
\text{have} \{xa \in A - \{x\}. (xa, x) \in R\} \subseteq A - \{x\} \text{ by blast} \\
\text{also have} \ldots \subseteq A \text{ using } 1 A \text{ by auto} \\
\text{finally have subset:} \{xa \in A - \{x\}. (xa, x) \in R\} \subset A . \\
\text{also have} \ldots \subseteq B \text{ by fact} \\
\text{finally have} ?X = \text{rqs-cost} (\text{card} \{xa \in A - \{x\}. (xa, x) \in R\}) \text{ using subset} \\
\text{by (intro psubset.III) auto} \\
\text{also have card} \{xa \in A - \{x\}. (xa, x) \in R\} = \text{linorder-rank R A x} \\
\text{by (simp add: linorder-rank-def)} \\
\text{finally have} ?X = \text{rqs-cost} . . . . . \} \\
\text{also} \{ \\
\text{have} \{xa \in A - \{x\}. (xa, x) \notin R\} \subseteq A - \{x\} \text{ by blast} \\
\text{also have} \ldots \subseteq A \text{ using } 1 A \text{ by auto} \\
\text{finally have subset:} \{xa \in A - \{x\}. (xa, x) \notin R\} \subset A . \\
\text{also have} \ldots \subseteq B \text{ by fact} \\
\text{finally have} ?Y = \text{rqs-cost} (\text{card} \{xa \in A - \{x\}. (xa, x) \notin R\}) \text{ using subset} \\
\text{by (intro psubset.III) auto} \\
\text{also} \{ \\
\text{have card} \{y \in A - \{x\}. (y, x) \in R\} \cup \{y \in A - \{x\}. (y, x) \notin R\} = \\
\text{linorder-rank R A x + card} \{xa \in A - \{x\}. (xa, x) \notin R\} \\
\text{unfolding linorder-rank-def using A by (intro card-UN-disjoint) auto} \\
\text{also have} \{y \in A - \{x\}. (y, x) \in R\} \cup \{y \in A - \{x\}. (y, x) \notin R\} = A - \{x\} \text{ by blast} \\
\text{also have card . . . = n using A 1 by (simp add: n-def)} \\
\text{finally have card} \{xa \in A - \{x\}. (xa, x) \notin R\} = n - \text{linorder-rank R A} \\
\text{A x by simp} \\
\text{finally have} ?Y = \text{rqs-cost} (n - \text{linorder-rank R A x}) . \} \\
\text{finally show} ?\text{case by (simp add: case-prod-unfold map-pmf-def)} \\
\text{qed} \\
\text{also have . . . = do} \{ \\
i \leftarrow \text{map-pmf} (\text{linorder-rank R A}) (\text{pmf-of-set} A); \\
(n1, n2) \leftarrow \text{pair-pmf} (\text{rqs-cost} i) (\text{rqs-cost} (n - i)); \\
\text{return-pmf} (n1 + n2) \\
\} \text{ by (simp add: bind-map-pmf)
also have map-pmf (linorder-rank R A) (pmf-of-set A) = pmf-of-set {..<card A}
by (intro map-pmf-of-set-bij-betw bij-betw-linorder-rank[OF assms(2)] A psubset.prems)
also from A have card A > 0 by (intro Nat.gr0I) auto
hence {..<card A} = {..n} by (auto simp: n-def)
also have map-pmf (λm. n + m) (do {i ← pmf-of-set {..n};
    (n1, n2) ← pair-pmf (rqs-cost i) (rqs-cost (n - i));
    return-pmf (n1 + n2)
}) = rqs-cost (Suc n)
by (simp add: pair-pmf-def map-bind-pmf case-prod-unfold bind-assoc-pmf bind-return-pmf add-ac)
also from A have card A > 0 by (intro Nat.gr0I) auto
hence Suc n = card A by (simp add: n-def)
finally show ?thesis .
qed

We therefore have the same expectation as well. (Note that we showed
rqs-cost-exp n = 2 * real (n + 1) * harm n - 4 * real n and rqs-cost-exp
∼[sequentially] (λx. 2 * real x * ln (real x)) before.
corollary expectation-qs-cost:
assumes finite A and linorder-on B R and A ⊆ B
defines random-list ≡ pmf-of-set (permutations-of-set A)
shows measure-pmf.expectation (map-pmf (qs-cost R) random-list) real =
  rqs-cost-exp (card A)
unfolding random-list-def
by (subst qs-cost-average-conv-rqs-cost[OF assms(1−3)]) (simp add: expectation-rqs-cost)

end

References