Cost Analysis of QuickSort

Manuel Eberl

April 20, 2020

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
   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
    Landau-Symbols.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 \Rightarrow 'a list \Rightarrow '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 \geq length xs \Longrightarrow delete-index n xs = xs
    by (simp add: delete-index-altdef)

lemma length-delete-index [simp]: n < length xs \Longrightarrow 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 \Longrightarrow 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 \Longrightarrow 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 \Longrightarrow n < length xs \Longrightarrow
    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 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])+
declare rquicksort.simps [simp del]

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

**lemma** set-pmf-of-set-lessThan-length [simp]:

\( xs \neq [] \implies \text{set-pmf} \ (\text{pmf-of-set} \ \{\ldots \less \text{length} \ \xs\}) = \{\ldots \less \text{length} \ \xs\} \)

\by (\text{subst set-pmf-of-set}) \text{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** rquicksort-correct:

\assumes \text{trans} \ R \ \text{and} \ \text{total-on} \ (\text{set} \ \xs) \ \text{R} \ \text{and} \ \forall x \in \text{set} \ \xs, \ (x, x) \in R

\assumes \ (ys, n) \in \text{set-pmf} \ (\text{rquicksort} \ R \ \xs)

\shows \ \text{sorted-wrt} \ R \ \ys \ \land \ \text{mset} \ \ys = \text{mset} \ \xs

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

\text{proof (induction-wrt \ } \ys \ \text{arbitrary:} \ \ys \ \text{n rule: length-induct)}

\text{case (1 \ } \xs)

\have \ IH: \ \text{sorted-wrt} \ R \ \xs \ \text{mset} \ \ys = \text{mset} \ \xs

\if \ (\xs, n) \in \text{set-pmf} \ (\text{rquicksort} \ R \ \ys) \ \text{length} \ \ys < \text{length} \ \xs \ \text{set} \ \ys \subseteq \text{set} \ \xs \ \text{for} \ \xs \ \ys \ \text{n}

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

\text{show ?case}

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

\case \ False

\text{with} \ 1.\text{prems} \ \text{obtain} \ \ls \ \rs \ n1 \ n2 \ i \ \text{where *:}

\ i < \text{length} \ \xs \ (\ls, n1) \in \text{set-pmf} \ (\text{rquicksort} \ R \ [y\leftarrow \text{delete-index} \ i \ \xs. \ (y, \ xs \ ! \ i) \in \ R])

\ (\rs, n2) \in \text{set-pmf} \ (\text{rquicksort} \ R \ [y\leftarrow \text{delete-index} \ i \ \xs. \ (y, \ xs \ ! \ i) \notin \ R])

\ys = \ls @ [\xs ! \ i] @ \rs

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

\text{note} \ \ys = 'ys = \ls @ [\xs ! \ i] @ \rs;

\define \ls' \text{where} \ \ls' = [y\leftarrow \text{delete-index} \ i \ \xs. \ (y, \ xs \ ! \ i) \in \ R]

\define \rs' \text{where} \ \rs' = [y\leftarrow \text{delete-index} \ i \ \xs. \ (y, \ xs \ ! \ i) \notin \ R]

\from \ i < \text{length} \ \xs \ \text{have less:} \ \text{length} \ \ls' < \text{length} \ \xs \ \text{length} \ \rs' < \text{length} \ \xs

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

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

\using * unfolding \ \ls'\text{-def} \ \rs'\text{-def by blast+}

\text{have subset: set} \ \ls' \subseteq \text{set} \ \xs \ \text{set} \ \rs' \subseteq \text{set} \ \xs

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

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

\text{have sorted:} \ \text{sorted-wrt} \ R \ \ls \ \text{sorted-wrt} \ R \ \rs

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

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

\text{have \ ls-le:} \ (x, \ xs \ ! \ i) \in \ R \ \text{if} \ x \in \text{set} \ \ls \ \text{for} \ x

\text{proof – }

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

\text{also note mset(1)}

\text{finally show \ ?thesis by (simp add: \ls'\text{-def}]}
have \( rs \text{-} qe : (x, xs ! i) \notin R \text{ if } x \in \text{ set } rs \text{ for } x \)
proof (induction \( x \))
  from that have \( x \in \# \text{ mset } rs \) by simp
  also note mset(2)
  finally have \( x : x \in \text{ set } rs' \) by simp
  thus \((x, xs ! i) \notin R \) by (simp-all add: \( rs' \text{-def} \))
  from \( x \) \and subset \and (\( i < \text{ length } xs \)) have \( x \in \text{ set } xs \)
  by auto
  with \( 1 \text{-prems and } ((x, xs ! i) \notin R) \) show \((xs ! i, x) \in R \)
  unfolding total-on-def by (cases xs ! i = x) auto
qed (insert 1.prems, simp-all)

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

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

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

theorem snd-rquicksort:
  assumes linorder-on A R and set xs \( \subseteq \) A and distinct xs
  shows \( \text{map-pmf } \text{snd } (rquicksort R xs) = \text{rqs-cost } (\text{length } xs) \)
  using assms(2-)
proof (induction xs rule: length-induct)
  case (1 xs)
  have IH: \( \text{map-pmf } \text{snd } (rquicksort R ys) = \text{rqs-cost } (\text{length } ys) \)
if length ys < length xs mset ys ⊆# mset xs for ys

proof –
  from set-mset-mono[of that(2)] have set ys ⊆ set xs by simp
  also note (set xs ⊆ A)
  ultimately have set ys ⊆ A.
  moreover from (distinct xs) and that(2) have distinct ys
    by (rule distinct-mset-mono)
  ultimately show thesis using that and 1.IH by blast
qed

define n where n = length xs
define cnt where cnt = (λi. length ys←index i xs. (y, xs ! i) ∈ R])
have cnt-altdef: cnt i = linorder-rank R (set xs) (xs ! i) if i: i < n for i
proof –
  have cnt i = length ys←index i xs. (y, xs ! i) ∈ R] by (simp add: cnt-def)
  also have ... = card (set ys←index i xs. (y, xs ! i) ∈ R])
    by (intro distinct-card [symmetric] distinct-filter distinct-delete-index 1.prems)
  also have set ys←index i xs. (y, xs ! i) ∈ R] =
    {x ∈ set xs−{xs!i}. (x, xs ! i) ∈ R}
    using 1.prems and i by (simp add: set-delete-index-distinct n-def)
  also have card ... = linorder-rank R (set xs) (xs ! i) by (simp add: linorder-rank-def)
  finally show thesis.
qed

from 1.prems have bij-betw (!!) xs {..<n} (set xs)
  by (intro bij-beta-byWitness[where f’ = index xs]) (auto simp: n-def index-nth-id)
moreover have bij-betw (linorder-rank R (set xs)) (set xs) {..<card (set xs)}
  using assms(1) by (rule bij-beta-linorder-rank) (insert 1.prems, auto)
ultimately have bij-betw (linorder-rank R (set xs) o (λi. xs ! i)) {..<n} {..<n}
  using 1.prems by (simp add: n-def o-def distinct-card)

show thesis

proof (cases xs = [])
  case False
  hence n > 0 by (simp add: n-def)
  hence [simp]: n ≠ 0 by (intro notI) auto
  from False have map-pmf snd (rqicksort R xs) =
    map-of-set {..<length xs} ∞ (
      λi. map-pmf (λz. length xs − 1 + fst z + snd z)
      (pair-pmf (map-pmf snd (rqicksort R [ys←index i xs. (y, xs
        ! i) ∈ R])))
      (map-pmf snd (rqicksort R [ys←index i xs. (y, xs ! i)
        ∉ R]))
    )
  )
    by (subst rqicksort.simps)
    (simp add: map-bind-pmf bind-map-pmf Let-def case-prod-unfold o-def
    pair-pmf-def)
also have \(\ldots = \text{pmf-of-set} \{..<\text{length } xs\} \gg\)
\[(\lambda i. \text{map-pmf} (\lambda z. n - 1 + \text{fst } z + \text{snd } z) (\text{pair-pmf} (\text{rqs-cost } (\text{cnt } i)) (\text{rqs-cost } (n - 1 - \text{cnt } i))))\)

proof (intro bind-pmf-cong refl, goal-cases)
case (1 i)
with \(\tau\) have \(i : i < \text{length } xs\) by auto
from \(i\) have \(\text{map-pmf } \text{snd} \ (\text{rquicksort } R \ [y\leftarrow\text{delete-index} \ i \ xs. \ (y, xs ! i) \notin R]) = \text{rqs-cost } (\text{length } [y\leftarrow\text{delete-index} \ i \ xs. \ (y, xs ! i) \notin R])\)
by (intro IH)
(auto intro!: le-less-trans[of length-filter-le] simp: mset-filter
  intro: subset-mset.order.trans multiset-filter-subset diff-subset-eq-self)
also have \(\text{length } [y\leftarrow\text{delete-index} \ i \ xs. \ (y, xs ! i) \notin R] = n - 1 - \text{cnt } i\)
unfolding n-def cnt-def
using sum-length-filter-compl[of \(\lambda y. (y, xs ! i) \in R \text{ delete-index } i \ xs\)] by simp
finally have \(\text{map-pmf } \text{snd} \ (\text{rquicksort } R \ [y\leftarrow\text{delete-index} \ i \ xs. \ (y, xs ! i) \notin R]) = \text{rqs-cost } (n - 1 - \text{cnt } i)\cdot\)
moreover have \(\text{map-pmf } \text{snd} \ (\text{rquicksort } R \ [y\leftarrow\text{delete-index} \ i \ xs. \ (y, xs ! i) \notin R]) = \text{rqs-cost } (\text{cnt } i)\)
unfolding cnt-def using \(i\)
by (intro IH)
(auto intro!: le-less-trans[of length-filter-le] simp: mset-filter
  intro: subset-mset.order.trans multiset-filter-subset diff-subset-eq-self)
ultimately show \(?case by (simp only: n-def)\)
qed
also have \(\ldots = \text{map-pmf } \text{cnt} \ (\text{pmf-of-set} \{..<n\}) \gg\)
\[(\lambda i. \text{map-pmf} (\lambda z. n - 1 + \text{fst } z + \text{snd } z) (\text{pair-pmf} (\text{rqs-cost } i) (\text{rqs-cost } (n - 1 - i))))\)
(is = bind-pmf - ?f) by (simp add: bind-map-pmf n-def)
also have \(\text{map-pmf } \text{cnt} \ (\text{pmf-of-set} \{..<n\}) = \)
\(\text{map-pmf} (\lambda x. \text{linorder-rank } R (\text{set } xs) (xs ! i)) \ (\text{pmf-of-set} \{..<n\})\)
using \(n > 0\) by (intro map-pmf-cong refl subst (asm) set-pmf-of-set) (auto simp: cnt-altdef)
also from \(n > 0\) have \(\ldots = \text{pmf-of-set} \{..<n\} \ by \ (\text{intro map-pmf-of-set-bij-betw bji}) \ auto\)
also have \(\text{pmf-of-set} \{..<n\} \gg\)
\(?f = \text{rqs-cost } n\)
by (cases n) (simp-all add: lessThan-Suc-atMost bind-map-pmf map-bind-pmf pair-pmf-def)
finally show \(?thesis by (simp add: n-def)\)
qed simp-all
qed

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-exp n) real = rqs-cost-exp n
proof (induction n rule: rqs-cost.induct)
case (Suc 2)
  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)))
      (λz. real (fst z) + real (snd z)) (is - = - + ?A)
    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)
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:

**Lemma** \( \text{rqs-cost-exp-Suc'} \):

\[
\text{rqs-cost-exp} \ (\text{Suc} \ n) = \text{real} \ n + 2 / \text{real} \ (\text{Suc} \ n) * (\sum i \leq n. \ \text{rqs-cost-exp} \ i)
\]

**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 \( \text{rqs-cost-exp-Suc} \))

- also have \( (\sum i \leq n. \ \text{rqs-cost-exp} \ i + \text{rqs-cost-exp} \ (n - i)) = (\sum i \leq n. \ \text{rqs-cost-exp} \ i) \)

  by (simp add: \text{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 \text{sum.reindex-bij-witness[of - } \lambda i. \ n - i \lambda i. \ n - i]) auto

- also have \( \ldots + \ldots = 2 \ldots \) by simp

- also have \( \ldots / \text{real} \ (\text{Suc} \ n) = 2 / \text{real} \ (\text{Suc} \ n) * (\sum i \leq n. \ \text{rqs-cost-exp} \ i) \) by simp

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

**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:

**Theorem** \( \text{rqs-cost-exp-eq} \) [code]: \( \text{rqs-cost-exp} \ n = 2 * \text{real} \ (n + 1) * \text{harm} \ n - 4 * \text{real} \ n \)

**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 \( F\)-Suc: \( F \ (\text{Suc} \ m) = F \ m + \text{real} \ (2*m) / (\text{real} \ ((m+1)*(m+2))) \) if \( m \geq 0 \) for \( m \)

  proof (cases \( m \))

  - case (Suc \( n \))

    have \( A: \text{rqs-cost-exp} \ (\text{Suc} \ (\text{Suc} \ n)) * \text{real} \ (\text{Suc} \ (\text{Suc} \ n)) = \)

    \( \text{real} \ ((n+1)*(n+2)) + 2 * (\sum i \leq n. \ \text{rqs-cost-exp} \ i) + 2 * \text{rqs-cost-exp} \)

    (\( \text{Suc} \ n \))

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

9
have B: \(\text{rqs-cost-exp} \ (\text{Suc} \ n) * \text{real} \ (\text{Suc} \ n) = \text{real} \ (n*(n+1)) + 2 * (\sum i \leq n. \ \text{rqs-cost-exp} \ i)\)
by (subst \(\text{rqs-cost-exp-Suc} \)) (simp-all add: field-simps)

have \(\text{rqs-cost-exp} \ (\text{Suc} \ (n+1)) * \text{real} \ (\text{Suc} \ (n+1)) - \text{rqs-cost-exp} \ (\text{Suc} \ n) * \text{real} \ (\text{Suc} \ n) = \)
\[\text{real} \ ((n+1)*(n+2)) - \text{real} \ (n*(n+1)) + 2 * \text{rqs-cost-exp} \ (\text{Suc} \ n)\]
by (subst A, subst B) simp-all
also have \(\text{real} \ ((n+1)*(n+2)) - \text{real} \ (n*(n+1)) = \text{real} \ (2*(n+1))\) by simp
finally have \(\text{rqs-cost-exp} \ (\text{Suc} \ (n+1)) * \text{real} \ (n+2) = \text{rqs-cost-exp} \ (\text{Suc} \ n) * \text{real} \ (n+3) + \text{real} \ (2*(n+1))\)
by (simp add: algebra-simps)


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

proof (cases \(n \geq 1\))
case True
thus \(?thesis\) by (simp add: F-def algebra-simps Suc)
qed simp-all

have \(F \ n = 2 * (\sum k=1..n. \ \text{real} \ (k - 1) / \text{real} \ (k * (k + 1)))\) (is - = 2 * ?S)
by (fact F-eq)
also have \(?S = (\sum k=1..n. \ 2 / \text{real} \ (\text{Suc} \ k) - 1 / \text{real} \ k)\)
by (intro sum.cong) (simp-all add: field-simps of-nat-diff)
also have \(= 2 * (\sum k=1..n. \ \text{inverse} \ (\text{real} \ (\text{Suc} \ k))) - \text{harm} \ n\)
by (subst sum-subtractf) (simp add: harm-def sum.distrib sum.distrib-left divide-simps)
also have \((\sum k=1..n. \ \text{inverse} \ (\text{real} \ (\text{Suc} \ k))) = (\sum k=\text{Suc} \ 1..\text{Suc} \ n. \ \text{inverse} \ (\text{real} \ k))\)
by (intro sum.reindex-bij-witness[OF - \(\lambda x. \ x < 1 \ \text{Suc}\)) auto
also have \(= \text{harm} \ (\text{Suc} \ n) - 1 \ \text{unfolding} \ \text{harm-def}\) by (subst (2) sum.atLeast-Suc-atMost simp-all)
finally have \(F \ n = 2 * \text{harm} \ n + 4 * (1 / (n + 1) - 1)\) by (simp add: harm-Suc field-simps)
also have \(= \) \(?thesis\) by (simp add: F-def add-ac)
finally show \(?thesis\) .
qed

lemma asymp-equiv-harm \(\text{asymp-equiv-intros}: \text{harm} \sim[\text{at-top}] (\lambda n. \ ln (\text{real} \ n))\)
proof -
have \((\lambda n. \ \text{harm} \ n - ln (\text{real} \ n)) \in O(\lambda- \ 1)\) using euler-mascheroni-LIMSEQ
by (intro bigo-freal-tends_to[where \(c = \text{euler-mascheroni}\)) simp-all
also have \((\lambda- \ 1) \in o(\lambda n. \ ln (\text{real} \ n))\) 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 \(\text{rqs-cost-exp-asymp-equiv}: \text{rqs-cost-exp} \sim_{\text{at-top}} (\lambda n. 2 * n * \ln n)\)
proof –
\text{have} \(\text{rqs-cost-exp} = (\lambda n. 2 * \text{real } (n + 1) * \text{harm } n - 4 * \text{real } n)\) using \(\text{rqs-cost-exp-eq} ..\)
also have \(\ldots = (\lambda n. 2 * \text{real } n * \text{harm } n + (2 * \text{harm } n - 4 * \text{real } n))\)
by (simp add: algebra-simps)
finally have \(\ldots \sim_{\text{at-top}} \ldots \) by simp
also have \(\ldots \sim_{\text{at-top}} (\lambda n. 2 * \text{real } n * \text{harm } n)\)
proof (subst asymp-equiv-add-right)
\text{have} \((\lambda x. 1 * \text{harm } x) \in o(\lambda x. \text{real } x * \text{harm } x)\)
by (intro landau-o-small-big-mult smallo-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 \((\lambda x. \text{real } x * 1) \in o(\lambda x. \text{real } x * \text{harm } x)\)
by (intro landau-o-big-small-mult) (simp-all add: smallomega-iff-smallo)
ultimately show \((\lambda n. 2 * \text{harm } n - 4 * \text{real } n) \in o(\lambda n. 2 * \text{real } n * \text{harm } n)\)
by (intro sum-in-smallo) simp-all
qed simp-all
also have \(\ldots \sim_{\text{at-top}} (\lambda n. 2 * \text{real } n * \ln (\text{real } n))\) by (intro asymp-equiv-intros)
finally show \(?\)thesis .
qed

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

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

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

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

finally show ?thesis by simp

qed auto

qed

lemma rqs-cost-exp-leI: \( m \leq n \Rightarrow rqs-cost-exp m \leq rqs-cost-exp n \)
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 (\( \lambda y. (y,x) \in R \wedge (x,y) \notin R \)) xs,
filter (\( \lambda y. (x,y) \in R \wedge (y,x) \notin R \)) xs,
filter (\( \lambda y. (x,y) \in R \wedge (y,x) \notin R \)) 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'.simsps [simp del]

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

context

begin

qualified definition lesss :: ('a × 'a) set ⇒ 'a ⇒ 'a list ⇒ 'a list where
lesss $R \cdot x \cdot xs$ = filter ($\lambda y. (y, x) \in R \wedge (x, y) \notin R$) $xs$

**qualified definition** greater :: ('a × 'a) set ⇒ 'a ⇒ 'a list ⇒ 'a list where
greater $R \cdot x \cdot xs$ = filter ($\lambda y. (x, y) \in R \wedge (y, x) \notin R$) $xs$

**qualified lemma** lesss-Con:
lesss $R \cdot x \cdot (y \# ys)$ =
(if $(y, x) \in R \wedge (x, y) \notin R$ then $y \#$ lesss $R \cdot x \cdot ys$ else lesss $R \cdot x \cdot ys$)
by (simp add: lesss-def)

**qualified lemma** length-lesss-le [intro]: length (lesss $R \cdot x \cdot xs$) ≤ length $xs$
by (simp add: lesss-def)

**qualified lemma** length-lesss-less [intro]:
assumes $x \in$ set $xs$
shows length (lesss $R \cdot x \cdot xs$) < length $xs$
using assms by (induction $xs$) (auto simp: lesss-Con intro: le-less-trans)

**qualified lemma** greater-Con:
greater $R \cdot x \cdot (y \# ys)$ =
(if $(x, y) \in R \wedge (y, x) \notin R$ then $y \#$ greater $R \cdot x \cdot ys$ else greater $R \cdot x \cdot ys$)
by (simp add: greater-def)

**qualified lemma** length-greater-ls [intro]: length (greater $R \cdot x \cdot xs$) ≤ length $xs$
by (simp add: greater-def)

**qualified lemma** length-greater-less [intro]:
assumes $x \in$ set $xs$
shows length (greater $R \cdot x \cdot xs$) < length $xs$
using assms by (induction $xs$) (auto simp: greater-Con intro: le-less-trans)

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

**function** rqs′-cost :: ('a × 'a) set ⇒ 'a ⇒ nat pmf where
rqs′-cost $R \cdot x \cdot xs$ =
(if $xs = []$ then return-pmf 0
else do
 i ← pmf-of-set {..<length $xs$};
 let $x = xs ! i$;
 map-pmf ($\lambda (n1, n2).$ length $xs - 1 + n1 + n2$)
 (pair-pmf (rqs′-cost $R \cdot (lesss \cdot R \cdot x \cdot xs)$) (rqs′-cost $R \cdot (greater \cdot R \cdot x \cdot xs)$))
)
by auto
termination by (relation Wellfounded.measure (length ∘ snd)) auto

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

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

lemma expectation-pair-pmf-fst [simp]:
fixes f :: 'a ⇒ 'b::{banach, second-countable-topology}
shows measure-pmf.expectation (pair-pmf p q) (λx. f (fst x)) = measure-pmf.expectation p f
proof -
have measure-pmf.expectation (pair-pmf p q) (λx. f (fst x)) = measure-pmf.expectation (map-pmf fst (pair-pmf p q)) f by simp
also have map-pmf fst (pair-pmf p q) = p by (simp add: map-fst-pair-pmf)
finally show thesis.
qed

lemma expectation-pair-pmf-snd [simp]:
fixes f :: 'a ⇒ 'b::{banach, second-countable-topology}
shows measure-pmf.expectation (pair-pmf p q) (λx. f (snd x)) = measure-pmf.expectation q f
proof -
have measure-pmf.expectation (pair-pmf p q) (λx. f (snd x)) = measure-pmf.expectation (map-pmf snd (pair-pmf p q)) f by simp
also have map-pmf snd (pair-pmf p q) = q by (simp add: map-snd-pair-pmf)
finally show thesis.
qed

qualified lemma length-lesss-le-sorted:
assumes sorted-wrt R xs i < length xs
shows length (lesss R (xs ! i) xs) ≤ i
using assms by (induction arbitrary: i rule: sorted-wrt.induct)
  (force simp: lessss-def nth-Cons le-Suc-eq split: nat.splits)+

qualified lemma length-greaters-le-sorted:
assumes sorted-wrt R xs i < length xs
shows length (greaters R (xs ! i) xs) ≤ 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 \subseteq A
shows \( \text{length} (\text{lesss} R (\text{insort-wrt} R \text{ !} \ i) \ \text{xs}) \leq i \)
proof
  -
  define \( x \) where \( x = \text{insort-wrt} R \text{ !} \ i \)
  define less where \( \lambda x \ y. \ (x, y) \in R \land (y, x) \notin R \)
  have \( \text{length} (\text{lesss} R x xs) = \text{size} \{ \# y \in\# \ \text{mset} \ \text{xs}. \ \text{less} y x \# \} \)
    by (simp add: lesss-def size-mset [symmetric] less-def mset-filter del: size-mset)
also have \( \text{mset} \ \text{xs} = \text{mset} (\text{insort-wrt} R \ \text{xs}) \) by simp
also have \( \text{size} \{ \# y \in\# \ \text{mset} (\text{insort-wrt} R \ \text{xs}). \ \text{less} y x \# \} = \text{length} (\text{lesss} R x (\text{insort-wrt} R \ \text{xs})) \)
  by (simp only: mset-filter [symmetric] size-mset lesss-def less-def)
also have \( \ldots \leq i \) unfolding x-def by (rule length-minus-less-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 \subseteq A
shows \( \text{length} (\text{greaters} R (\text{insort-wrt} R \ \text{!} \ i) \ \text{xs}) \leq \text{length} \ \text{xs} - i - 1 \)
proof
  -
  define \( x \) where \( x = \text{insort-wrt} R \ \text{xs} \ ! \ i \)
  define less where \( \lambda x \ y. \ (x, y) \in R \land (y, x) \notin R \)
  have \( \text{length} (\text{greaters} R x xs) = \text{size} \{ \# y \in\# \ \text{mset} \ \text{xs}. \ \text{less} x y \# \} \)
    by (simp add: greaters-def size-mset [symmetric] less-def mset-filter del: size-mset)
also have \( \text{mset} \ \text{xs} = \text{mset} (\text{insort-wrt} R \ \text{xs}) \) by simp
also have \( \text{size} \{ \# y \in\# \ \text{mset} (\text{insort-wrt} R \ \text{xs}). \ \text{less} x y \# \} = \text{length} (\text{greaters} R x (\text{insort-wrt} R \ \text{xs})) \)
  by (simp only: mset-filter [symmetric] size-mset greaters-def less-def)
also have \( \ldots \leq \text{length} (\text{insort-wrt} R \ \text{xs}) - i - 1 \) 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 \subseteq A
shows measure-pmf.expectation (rqs'-cost R xs) real \leq rqs-cost-exp (length xs)
using assms
proof (induction R xs rule: rqs'-cost.induct)
case (1 R xs)
show ?case
proof (cases xs = [])

15
case False

define n where $n = \text{length } xs - 1$

have $\text{length-eq}: \text{length } xs = \text{Suc } n$ using False by (simp add: n-def)

define $E$ where $E = (\lambda xs. \text{measure-pmf.expectation } (\text{rqs'}-\text{cost } R xs) \text{ real})$

define $f$ where $f = (\lambda x. \text{rqs-cost-exp } (\text{less } R x \text{ xs}) + \text{rqs-cost-exp } (\text{length } (\text{great } R x \text{ xs})))$

have $\text{rqs'}-\text{cost } R xs =$
  do {  
    $i \leftarrow \text{pmf-of-set } \{..<\text{length } xs\};$
    $\text{map-pmf } (\lambda (n1, y). \text{length } xs - \text{Suc } 0 + n1 + y)$
    $\text{(pair-pmf } (\text{rqs'}-\text{cost } R (\text{less } R (xs ! i) xs))$
    $\text{ (rqs'}-\text{cost } R (\text{great } R (xs ! i) xs)))$
  }  
using False by (subt rqs'-cost.simps) (simp-all add: Let-def)

also have $\text{measure-pmf.expectation } \ldots \text{ real } = \text{ real } n + (\sum k<\text{length } xs. E (\text{less } R (xs ! k) xs) + E (\text{great } R (xs ! k) xs)) / \text{real (length } xs)$
using False
by (subt pmf-expectation-bind-pmf-of-set)

also have $\ldots \leq \text{ real } n + (\sum k<\text{length } xs. f (xs ! k)) / \text{real (length } xs)$

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 $(\sum k<\text{length } xs. f (xs ! k)) = (\sum x \in \#\text{mset } xs. f x)$
by (simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth)
(simp-all add: atLeast0LessThan)

also have $\text{mset } xs = \text{mset } (\text{insort-wrt } R xs)$
by simp

also have $(\sum x \in \#. \ldots. f x) = (\sum i<\text{length } xs. f (\text{insort-wrt } R xs ! i))$
by (simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth)
(simp-all add: atLeast0LessThan)

also have $\ldots \leq (\sum i<\text{length } xs. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (\text{length } xs - i - 1))$

unfolding f-def

proof (intro sum-mono add-mono rqs-cost-exp-le1)
fix i assume i: $i \in \{..<\text{length } xs\}$
show $\text{length } (\text{less } R (\text{insort-wrt } R \text{ xs ! i} \text{ xs}) \leq i$
  using i 1.prems by (intro length-less-le'[where $A = A$]) auto
show $\text{length } (\text{great } R (\text{insort-wrt } R \text{ xs ! i} \text{ xs}) \leq \text{length } xs - i - 1$
  using i 1.prems by (intro length-greaters-le'[where $A = A$]) auto
qed

also have $\ldots = (\sum i \leq n. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i))$
by (intro sum.cong) (auto simp: length-eq)

also have $\text{real } n + \ldots / \text{real (length } xs) = \text{rqs-cost-exp } (\text{length } xs)$
by (simp add: length-eq rqs-cost-exp.simps(2))
2 Average case analysis of deterministic Quick-Sort

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 \( \forall x. x \in set xs \Rightarrow (x, x) \in R \)
  shows sorted-wrt R (quicksort R xs)
  using assms
  proof (induction R xs rule: quicksort.induct)
    case (2 R x ys)
    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) \notin 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
case False
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 (length xs + qs-cost R [y ← xs. (y,x) ∈ R] + qs-cost R [y ← xs. (y,x) ∉ R])
}
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})))
\( \trianglerighteq (\lambda (ls, rs). \text{return-pmf} (\text{qs-cost} R \, ls + \text{qs-cost} R \, rs)) = \) 
\[
\text{map-pmf} (\lambda (n1, n2). n1 + n2) (\text{pair-pmf} \\
(\text{map-pmf} (\text{qs-cost} R) (\text{pmf-of-set} (\text{permutations-of-set} \{ xa \in A - \{ x \}. (xa, x) \in R \}))) \\
(\text{map-pmf} (\text{qs-cost} R) (\text{pmf-of-set} (\text{permutations-of-set} \{ xa \in A - \{ x \}. (xa, x) \notin R \}))) \\
(\text{is} - \text{= map-pmf} - (\text{pair-pmf} ?X \, ?Y)) \\
\) by (\text{subst partition-random-permutations}) 
\] 
\( \) (\text{simp-all add: map-pmf-def case-prod-unfold bind-return-pmf bind-assoc-pmf} pair-pmf-def A) 
\( \) also \{ 
\( \) 
\( \) have \{ \{ xa \in A - \{ x \}. (xa, x) \in R \} \subseteq A - \{ x \} \) by blast 
\( \) also have \ldots \( \subseteq A \) using \( I \, A \) by auto 
\( \) finally have \text{subset}: \{ xa \in A - \{ x \}. (xa, x) \in R \} \subseteq A \. 
\( \) also have \ldots \( \subseteq B \) by fact 
\( \) finally have \( ?X = \text{rqs-cost} (\text{card} \, \{ xa \in A - \{ x \}. (xa, x) \in R \}) \) using \( \text{subset} \) 
\( \) by (\text{intro psubset.\( I \).I}) auto 
\( \) also have \( \text{card} \, \{ xa \in A - \{ x \}. (xa, x) \in R \} = \text{linorder-rank} R \, A \, x \) 
\( \) by (\text{simp add: linorder-rank-def}) 
\( \) finally have \( ?X = \text{rqs-cost} \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \) 
\} 
\( \) also \{ 
\( \) 
\( \) have \{ \{ xa \in A - \{ x \}. (xa, x) \notin R \} \subseteq A - \{ x \} \) by blast 
\( \) also have \ldots \( \subseteq A \) using \( I \, A \) by auto 
\( \) finally have \text{subset}: \{ xa \in A - \{ x \}. (xa, x) \notin R \} \subseteq A \. 
\( \) also have \ldots \( \subseteq B \) by fact 
\( \) finally have \( ?Y = \text{rqs-cost} (\text{card} \, \{ xa \in A - \{ x \}. (xa, x) \notin R \}) \) using \( \text{subset} \) 
\( \) by (\text{intro psubset.\( I \).I}) auto 
\( \) also \{ 
\( \) 
\( \) have \( \text{card} \) \ldots \( = n \) using \( A \) I by (\text{simp add: n-def}) 
\( \) finally have \( \text{card} \, \{ xa \in A - \{ x \}. (xa, x) \notin R \} = n - \text{linorder-rank} R \, A \, x \) by \text{simp} 
\} 
\( \) finally have \( ?Y = \text{rqs-cost} (n - \text{linorder-rank} R \, A \, x) \) 
\} 
\( \) finally show \( \) \text{case} by (\text{simp add: case-prod-unfold map-pmf-def}) 
\( \text{qed} \) 
\( \) also have \ldots \( = \) do \{ 
\( \) 
\( \) \( i \leftarrow \text{map-pmf} (\text{linorder-rank} R \, A) (\text{pmf-of-set} A); \) 
\( \) \( (n1, n2) \leftarrow \text{pair-pmf} (\text{rqs-cost} \, i) (\text{rqs-cost} \, (n - i)); \) 
\( \) \( \text{return-pmf} (n1 + n2) \) 
\} by (\text{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