Cost Analysis of QuickSort

Manuel Eberl

September 23, 2023

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 $2\,n \ln n$; 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

theory *Randomised-Quick-Sort*
imports
HOL−Probability.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* ⇒ *'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 (λ*y*. (*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 *set-pmf-of-set-lessThan-length* [*simp*]:
  $xs \neq [] \Longrightarrow$ *set-pmf* (*pmf-of-set* {..<*length xs*}) = {..<*length xs*}
  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 *rquicksort-correct*:
  assumes *trans R* **and** *total-on* (*set xs*) *R* **and** $\forall x \in$*set xs*. (*x*,*x*) $\in$ *R*
  assumes (*ys*, *n*) $\in$ *set-pmf* (*rquicksort R xs*)
  shows *sorted-wrt R ys* $\wedge$ *mset ys* = *mset xs*
  using *assms*(*2*−)
proof (*induction xs arbitrary*: *ys n rule*: *length-induct*)
  case (*1 xs*)
  have *IH*: *sorted-wrt R zs mset zs* = *mset ys*
    if (*zs*, *n*) $\in$ *set-pmf* (*rquicksort R ys*) *length ys* < *length xs set ys* $\subseteq$ *set xs* **for** *zs ys n*
    using *that 1.IH total-on-subset*[*OF 1.prems*(*1*) *that*(*3*)] *1.prems*(*2*) **by** *blast*+
  show *?case*
  proof (*cases xs* = [])
    case *False*
    with *1.prems* **obtain** *ls rs n1 n2 i* **where** ∗:
      i < *length xs* (*ls*, *n1*) $\in$ *set-pmf* (*rquicksort R* [*y*←*delete-index i xs*. (*y*, *xs* ! *i*) $\in$ *R*])
      (*rs*, *n2*) $\in$ *set-pmf* (*rquicksort R* [*y*←*delete-index i xs*. (*y*, *xs* ! *i*) $\notin$ *R*])
      ys = *ls* @ [*xs* ! *i*] @ *rs*
      by (*subst* (*asm*) *rquicksort.simps*[*of* - *xs*]) (*auto simp*: *Let-def o-def*)
    note *ys* = ‹*ys* = *ls* @ [*xs* ! *i*] @ *rs*›
    define *ls′* **where** *ls′* = [*y*←*delete-index i xs*. (*y*, *xs* ! *i*) $\in$ *R*]
    define *rs′* **where** *rs′* = [*y*←*delete-index i xs*. (*y*, *xs* ! *i*) $\notin$ *R*]
    from ‹*i* < *length xs*› **have** *less*: *length ls′* < *length xs length rs′* < *length xs*
      unfolding *ls′-def rs′-def* **by** (*intro le-less-trans*[*OF length-filter-le*]; *force*)+
     have *ls*: (*ls*, *n1*) $\in$ *set-pmf* (*rquicksort R ls′*) **and** *rs*: (*rs*, *n2*) $\in$ *set-pmf* (*rquicksort R rs′*)
      using ∗ **unfolding** *ls′-def rs′-def* **by** *blast*+
    have *subset*: *set ls′* $\subseteq$ *set xs set rs′* $\subseteq$ *set xs*
      using *insert-set-delete-index*[*of i xs*] ‹*i* < *length xs*›
      by (*auto simp*: *ls′-def rs′-def*)
    have *sorted*: *sorted-wrt R ls sorted-wrt R rs*
     and *mset*: *mset ls* = *mset ls′ mset rs* = *mset rs′*
      by (*rule IH*[*of ls n1 ls′*] *IH*[*of rs n2 rs′*] *less ls rs subset*)+

    have *ls-le*: (*x*, *xs* ! *i*) $\in$ *R* **if** *x* $\in$ *set ls* **for** *x*
    proof −
      from *that* **have** *x* ∈# *mset ls* **by** *simp*
      also note *mset*(*1*)
      finally show *?thesis* **by** (*simp add*: *ls′-def*)

```
    qed
    have rs-ge: (x, xs ! i) ∉ R (xs ! i, x) ∈ R if x ∈ set rs for x
    proof −
      from that have x ∈# mset rs by simp
      also note mset(2)
      finally have x: x ∈ set rs′ by simp
      thus (x, xs ! i) ∉ R by (simp-all add: rs′-def)
       from x and subset and ‹i < length xs› have x ∈ set xs xs ! i ∈ set xs by
auto
      with 1.prems and ‹(x, xs ! i) ∉ R› show (xs ! i, x) ∈ R
        unfolding total-on-def by (cases xs ! i = x) auto
    qed

    have sorted-wrt R ys unfolding ys
      by (intro sorted-wrt-append ‹trans R› sorted-wrt-singleton sorted)
         (auto intro: rs-ge ls-le transD[OF ‹trans R›, of - xs!i])
    moreover have mset ys = mset xs unfolding ys using ‹i < length xs›
      by (simp add: mset ls′-def rs′-def add-mset-delete-index)
    ultimately show ?thesis ..
  qed (insert 1.prems, 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 rqs-cost :: nat ⇒ nat pmf where
  rqs-cost 0 = return-pmf 0
| rqs-cost (Suc n) =
    do {i ← pmf-of-set {..n}; a ← rqs-cost i; b ← rqs-cost (n − i); 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-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-cost* for the length of that list.

```
theorem snd-rquicksort:
  assumes linorder-on A R and set xs ⊆ A and distinct xs
  shows   map-pmf snd (rquicksort R xs) = rqs-cost (length xs)
  using assms(2−)
proof (induction xs rule: length-induct)
  case (1 xs)
  have IH: map-pmf snd (rquicksort R ys) = rqs-cost (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›
  finally 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 [y←delete-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 [y←delete-index i xs. (y, xs ! i) ∈ R] by (simp add: cnt-def)
  also have ... = card (set [y←delete-index i xs. (y, xs ! i) ∈ R])
   by (intro distinct-card [symmetric] distinct-filter distinct-delete-index 1.prems)
  also have set [y←delete-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-betw-byWitness[where f' = index xs]) (auto simp: n-def in-
dex-nth-id)
 moreover have bij-betw (linorder-rank R (set xs)) (set xs) {..<card (set xs)}
  using assms(1) by (rule bij-betw-linorder-rank) (insert 1.prems, auto)
 ultimately have bij-betw (linorder-rank R (set xs) ∘ (λi. xs ! i)) {..<n} {..<card
(set xs)}
  by (rule bij-betw-trans)
 hence bij: bij-betw (λi. linorder-rank R (set xs) (xs ! i)) {..<n} {..<n}
  using 1.prems by (simp add: n-def o-def distinct-card)

 show ?case
 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 (rquicksort R xs) =
          pmf-of-set {..<length xs} ⤜
           (λi. map-pmf (λz. length xs − 1 + fst z + snd z)
             (pair-pmf (map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs !
i) ∈ R]))
                       (map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i)
∉ R]))))
    by (subst rquicksort.simps)
       (simp add: map-bind-pmf bind-map-pmf Let-def case-prod-unfold o-def
pair-pmf-def)
```

```
  also have ... = pmf-of-set {..<length xs} ⤜
                  (λi. map-pmf (λz. n − 1 + fst z + snd z)
                    (pair-pmf (rqs-cost (cnt i)) (rqs-cost (n − 1 − cnt i))))
  proof (intro bind-pmf-cong refl, goal-cases)
    case (1 i)
    with ‹xs ≠ []› have i: i < length xs by auto
     from i have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i) ∉
R]) =
                          rqs-cost (length [y←delete-index i xs. (y, xs ! i) ∉ R])
      by (intro IH)
        (auto intro!: le-less-trans[OF length-filter-le]
              intro: subset-mset.trans multiset-filter-subset diff-subset-eq-self)
     also have length [y←delete-index i xs. (y, xs ! i) ∉ R] = n − 1 − cnt i
      unfolding n-def cnt-def
      using sum-length-filter-compl[of λy. (y, xs ! i) ∈ R delete-index i xs] i by
simp
     finally have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i) ∉
R]) =
                  rqs-cost (n − 1 − cnt i) .
     moreover have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i)
∈ R]) =
                   rqs-cost (cnt i) unfolding cnt-def using i
      by (intro IH)
        (auto intro!: le-less-trans[OF length-filter-le]
              intro: subset-mset.trans multiset-filter-subset diff-subset-eq-self)
     ultimately show ?case by (simp only: n-def)
  qed
  also have ... = map-pmf cnt (pmf-of-set {..<n}) ⤜
       (λi. map-pmf (λz. n − 1 + fst z + snd z) (pair-pmf (rqs-cost i) (rqs-cost
(n − 1 − i))))
     (is - = bind-pmf - ?f) by (simp add: bind-map-pmf n-def)
  also have map-pmf cnt (pmf-of-set {..<n}) =
              map-pmf (λi. linorder-rank R (set xs) (xs ! i)) (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 ... = pmf-of-set {..<n} by (intro map-pmf-of-set-bij-betw
bij) auto
  also have pmf-of-set {..<n} ⤜ ?f = 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 *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* ⤜ (λ*aa*. *rqs-cost* (*n* − *a*) ⤜
(λ*b*. *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* ⤜ (λ*a*. *rqs-cost* (*n* − *i*) ⤜ (λ*b*. *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*)))
(λ*z*. *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*)))
(λ*z*. *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*)
finally show *?case* **by** *simp*

```
  qed
  also have ... = (∑ i≤n. inverse (real (Suc n)) * real n) +
                  (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n − i)) / real (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 (∑ i≤n. inverse (real (Suc n)) * real n) = real n by simp
  also have ... + (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n − i)) / real (Suc n) =
rqs-cost-exp (Suc n)
    by (simp add: rqs-cost-exp-Suc)
  finally show ?case .
qed simp-all
```

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

```
lemma rqs-cost-exp-Suc':
  rqs-cost-exp (Suc n) = real n + 2 / real (Suc n) * (∑ i≤n. rqs-cost-exp i)
proof −
  have rqs-cost-exp (Suc n) = real n + (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n −
i)) / real (Suc n)
    by (rule rqs-cost-exp-Suc)
  also have (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n − i)) = (∑ i≤n. rqs-cost-exp
i) + (∑ i≤n. rqs-cost-exp (n − i))
    by (simp add: sum.distrib)
  also have (∑ i≤n. rqs-cost-exp (n − i)) = (∑ i≤n. rqs-cost-exp i)
    by (intro sum.reindex-bij-witness[of - λi. n − i λi. n − i]) auto
  also have ... + ... = 2 * ... by simp
  also have ... / real (Suc n) = 2 / real (Suc n) * (∑ i≤n. 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:

```
theorem rqs-cost-exp-eq [code]: rqs-cost-exp n = 2 * real (n + 1) * harm n − 4
* real n
proof −
  define F where F = (λn. rqs-cost-exp n / (real n + 1))
  have [simp]: F 0 = 0 F (Suc 0) = 0 by (simp-all add: F-def)
  have F-Suc: F (Suc m) = F m + real (2*m) / (real ((m+1)*(m+2))) if m >
0 for m
  proof (cases m)
    case (Suc n)
    have A: rqs-cost-exp (Suc (Suc n)) * real (Suc (Suc n)) =
            real ((n+1)*(n+2)) + 2 * (∑ i≤n. rqs-cost-exp i) + 2 * rqs-cost-exp
(Suc n)
      by (subst rqs-cost-exp-Suc') (simp-all add: field-simps)
    have B: rqs-cost-exp (Suc n) * real (Suc n) = real (n*(n+1)) + 2 * (∑ i≤n.
rqs-cost-exp i)
```

```
    by (subst rqs-cost-exp-Suc') (simp-all add: field-simps)
  have rqs-cost-exp (Suc (Suc n)) * real (Suc (Suc n)) - rqs-cost-exp (Suc n) *
real (Suc n) =
        real ((n+1)*(n+2)) - real (n*(n+1)) + 2 * 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 rqs-cost-exp (Suc (Suc n)) * real (n+2) = rqs-cost-exp (Suc n) *
real (n+3) + real (2*(n+1))
    by (simp add: algebra-simps)
  hence rqs-cost-exp (Suc (Suc n)) / real (n+3) =
       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 ?thesis by (simp add: F-def algebra-simps Suc)
 qed simp-all

 have F-eq: F n = 2 * (∑ k=1..n. real (k - 1) / real (k * (k + 1))) for n
 proof (cases n ≥ 1)
  case True
   thus ?thesis by (induction n rule: dec-induct) (simp-all add: F-Suc alge-
bra-simps)
 qed (simp-all add: not-le)

 have F n = 2 * (∑ k=1..n. real (k - 1) / real (k * (k + 1))) (is - = 2 * ?S)
by (fact F-eq)
 also have ?S = (∑ k=1..n. 2 / real (Suc k) - 1 / real k)
   by (intro sum.cong) (simp-all add: field-simps of-nat-diff)
 also have ... = 2 * (∑ k=1..n. inverse (real (Suc k))) - harm n
   by (subst sum-subtractf) (simp add: harm-def sum.distrib sum-distrib-left di-
vide-simps)
 also have (∑ k=1..n. inverse (real (Suc k))) = (∑ k=Suc 1..Suc n. inverse (real
k))
   by (intro sum.reindex-bij-witness[of - λx. x - 1 Suc]) auto
 also have ... = harm (Suc n) - 1 unfolding harm-def by (subst (2) sum.atLeast-Suc-atMost)
simp-all
 finally have F n = 2 * harm n + 4 * (1 / (n + 1) - 1) by (simp add: harm-Suc
field-simps)
 also have ... * real (n + 1) = 2 * real (n + 1) * harm n - 4 * real n
   by (simp add: field-simps)
 also have F n * real (n + 1) = rqs-cost-exp n by (simp add: F-def add-ac)
 finally show ?thesis .
qed


lemma asymp-equiv-harm [asymp-equiv-intros]: harm ∼[at-top] (λn. ln (real n))
proof -
 have (λn. harm n - ln (real n)) ∈ O(λ-. 1) using euler-mascheroni-LIMSEQ
   by (intro bigoI-tendsto[where c = euler-mascheroni]) simp-all
 also have (λ-. 1) ∈ o(λn. ln (real n)) by auto
```

```
  finally have (λn. ln (real n) + (harm n − ln (real n))) ∼[at-top] (λn. ln (real
n))
    by (subst asymp-equiv-add-right) simp-all
  thus ?thesis by simp
qed

corollary rqs-cost-exp-asymp-equiv: rqs-cost-exp ∼[at-top] (λn. 2 ∗ n ∗ ln n)
proof −
   have rqs-cost-exp = (λn. 2 ∗ real (n + 1) ∗ harm n − 4 ∗ real n) using
rqs-cost-exp-eq ..
  also have ... = (λn. 2 ∗ real n ∗ harm n + (2 ∗ harm n − 4 ∗ real n))
    by (simp add: algebra-simps)
  finally have rqs-cost-exp ∼[at-top] ... by simp
  also have ... ∼[at-top] (λn. 2 ∗ real n ∗ harm n)
  proof (subst asymp-equiv-add-right)
    have (λx. 1 ∗ harm x) ∈ o(λx. real x ∗ harm x)
      by (intro landau-o.small-big-mult smallo-real-nat-transfer) simp-all
    moreover have harm ∈ ω(λ-. 1 :: real)
      by (intro smallomegaI-filterlim-at-top-norm) (auto simp: harm-at-top)
    hence (λx. real x ∗ 1) ∈ o(λx. real x ∗ harm x)
      by (intro landau-o.big-small-mult) (simp-all add: smallomega-iff-smallo)
    ultimately show (λn. 2 ∗ harm n − 4 ∗ real n) ∈ o(λn. 2 ∗ real n ∗ harm n)
      by (intro sum-in-smallo) simp-all
  qed simp-all
 also have ... ∼[at-top] (λn. 2 ∗ real n ∗ ln (real n)) by (intro asymp-equiv-intros)
  finally show ?thesis .
qed

lemma harm-mono: m ≤ n ⟹ harm m ≤ (harm n :: real)
  unfolding harm-def by (intro sum-mono2) auto

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

lemma harm-ge-1: n > 0 ⟹ harm n ≥ (1::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 ≤ rqs-cost-exp (Suc n)
  proof (cases n = 0)
    case False
    have 0 < (1 ∗ 2 ∗ (real n + 1) − 2 ∗ real n) / (real n + 1) by simp
   also have ... ≤ (harm n ∗ 2 ∗ (real n + 1) − 2 ∗ real n) / (real n + 1) using
False
      by (intro divide-right-mono diff-right-mono mult-right-mono) (auto simp:
harm-ge-1)
    also have ... = 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
qed

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 ∘ 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] simp: threeway-partition-def)+

declare rquicksort'.simps [simp del]

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

context
begin

qualified definition lesss :: ('a × 'a) set ⇒ 'a ⇒ 'a list ⇒ 'a list where
  lesss R x xs = filter (λy. (y, x) ∈ R ∧ (x, y) ∉ R) xs
```

qualified definition *greaters* :: ('*a* × '*a*) *set* ⇒ '*a* ⇒ '*a list* ⇒ '*a list* **where**
greaters R x xs = *filter* (λ*y*. (*x*, *y*) ∈ *R* ∧ (*y*, *x*) ∉ *R*) *xs*

qualified lemma *lesss-Cons*:
lesss R x (*y* # *ys*) =
(*if* (*y*, *x*) ∈ *R* ∧ (*x*, *y*) ∉ *R then y* # *lesss R x ys else lesss R x ys*)
by (*simp add*: *lesss-def*)

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

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

qualified lemma *greaters-Cons*:
greaters R x (*y* # *ys*) =
(*if* (*x*, *y*) ∈ *R* ∧ (*y*, *x*) ∉ *R then y* # *greaters R x ys else greaters R x ys*)
by (*simp add*: *greaters-def*)

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

qualified lemma *length-greaters-less* [*intro*]:
assumes *x* ∈ *set xs*
shows *length* (*greaters R x xs*) < *length xs*
using *assms* **by** (*induction xs*) (*auto simp*: *greaters-Cons intro*: *le-less-trans*)

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

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

declare *rqs′-cost.simps* [*simp del*]

lemma *rqs′-cost-nonempty*:

```
  xs ≠ [] ⟹ rqs'-cost R xs =
    do {
      i ← pmf-of-set {..<length xs};
      let x = xs ! i;
      n1 ← rqs'-cost R (lesss R x xs);
      n2 ← rqs'-cost R (greaters R x xs);
      return-pmf (length xs − 1 + n1 + n2)
    }
  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: lesss-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 ⊆ 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)
      (auto intro!: finite-imageI finite-cartesian-product simp: case-prod-unfold
        integrable-measure-pmf-finite sum-divide-distrib [symmetric] field-simps
        length-eq sum.distrib E-def)
  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: lesss-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 i 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 i 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))
  finally show ?thesis by (simp add: divide-right-mono)
```

qed (*auto simp*: *rqs′-cost.simps*)
qed

end
end

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* (λ*y*. (*y*,*x*) ∈ *R*) *xs*) @ [*x*] @ *quicksort R* (*filter* (λ*y*. (*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** ⋀*x*. *x* ∈ *set xs* ⟹ (*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* (λ*y*. (*y*,*x*) ∈ *R*) *xs*))
sorted-wrt R (*quicksort R* (*filter* (λ*y*. (*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* $\subseteq$ *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* $\subseteq$ *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])
                } 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})))
              ⤜ (λ(ls, rs). return-pmf (qs-cost R ls + qs-cost R rs)) =
            map-pmf (λ(n1, n2). n1 + n2) (pair-pmf
```

```
          (map-pmf (qs-cost R) (pmf-of-set (permutations-of-set {xa ∈ A − {x}.
(xa, x) ∈ R})))
          (map-pmf (qs-cost R) (pmf-of-set (permutations-of-set {xa ∈ A − {x}.
(xa, x) ∉ R}))))
       (is - = map-pmf - (pair-pmf ?X ?Y))
       by (subst partition-random-permutations)
       (simp-all add: map-pmf-def case-prod-unfold bind-return-pmf bind-assoc-pmf
pair-pmf-def A)
     also {
       have {xa ∈ A − {x}. (xa, x) ∈ R} ⊆ A − {x} by blast
       also have ... ⊂ A using 1 A by auto
       finally have subset: {xa ∈ A − {x}. (xa, x) ∈ R} ⊂ A .
       also have ... ⊆ B by fact
        finally have ?X = rqs-cost (card {xa ∈ A − {x}. (xa, x) ∈ R}) using
subset
         by (intro psubset.IH) auto
       also have card {xa ∈ A − {x}. (xa, x) ∈ R} = linorder-rank R A x
         by (simp add: linorder-rank-def)
       finally have ?X = rqs-cost ... .
     }
     also {
       have {xa ∈ A − {x}. (xa, x) ∉ R} ⊆ A − {x} by blast
       also have ... ⊂ A using 1 A by auto
       finally have subset: {xa ∈ A − {x}. (xa, x) ∉ R} ⊂ A .
       also have ... ⊆ B by fact
        finally have ?Y = rqs-cost (card {xa ∈ A − {x}. (xa, x) ∉ R}) using
subset
         by (intro psubset.IH) auto
       also {
         have card ({y∈A−{x}. (y,x)∈R} ∪ {y∈A−{x}. (y,x)∉R}) =
                linorder-rank R A x + card {xa ∈ A − {x}. (xa, x) ∉ R}
           unfolding linorder-rank-def using A by (intro card-Un-disjoint) auto
         also have {y∈A−{x}. (y,x)∈R} ∪ {y∈A−{x}. (y,x)∉R} = A − {x} by
blast
         also have card ... = n using A 1 by (simp add: n-def)
         finally have card {xa ∈ A − {x}. (xa, x) ∉ R} = n − linorder-rank R A
x by simp
       }
       finally have ?Y = rqs-cost (n − linorder-rank R A x) .
     }
     finally show ?case by (simp add: case-prod-unfold map-pmf-def)
   qed
   also have ... = do {
                     i ← map-pmf (linorder-rank R A) (pmf-of-set A);
                     (n1, n2) ← pair-pmf (rqs-cost i) (rqs-cost (n − i));
                     return-pmf (n1 + n2)
                   } 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* (λ*m*. *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
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*] (λ*x*. *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

[1] J. Cichoń. Quick Sort – average complexity.

[2] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10, 1962.

[3] R. Sedgewick. The analysis of Quicksort programs. *Acta Inf.*, 7(4):327–355, Dec. 1977.