

Cost Analysis of QuickSort

Manuel Eberl

February 6, 2026

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.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:  $n \geq \text{length } xs \implies \text{delete-index } n \text{ } xs = xs$ 
by (simp add: delete-index-altdef)
```

```
lemma length-delete-index [simp]:  $n < \text{length } xs \implies \text{length } (\text{delete-index } n \text{ } xs) =$ 
 $\text{length } xs - 1$ 
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:
   $n < \text{length } xs \implies \text{insert } (xs ! n) (\text{set } (\text{delete-index } n \text{ } xs)) = \text{set } xs$ 
by (induction n xs rule: delete-index.induct) auto
```

```
lemma add-mset-delete-index:
   $i < \text{length } xs \implies \text{add-mset } (xs ! i) (\text{mset } (\text{delete-index } i \text{ } xs)) = \text{mset } xs$ 
by (induction i xs rule: delete-index.induct) simp-all
```

```
lemma nth-delete-index:
   $i < \text{length } xs \implies n < \text{length } xs \implies$ 
   $\text{delete-index } n \text{ } xs ! i = (\text{if } i < n \text{ then } xs ! i \text{ else } xs ! \text{Suc } i)$ 
by (auto simp: delete-index-altdef nth-append min-def)
```

```
lemma set-delete-index-distinct:
assumes distinct xs  $n < \text{length } xs$ 
shows  $\text{set } (\text{delete-index } n \text{ } xs) = \text{set } xs - \{xs ! n\}$ 
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]:
   $i < \text{length } xs \implies \text{mset } (\text{delete-index } i \text{ } xs) = \text{mset } xs - \{\# \text{ } xs!i \#\}$ 
  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 *set-pmf-of-set-lessThan-length* [simp]:
 $xs \neq [] \implies \text{set-pmf } (\text{pmf-of-set } \{..<\text{length } xs\}) = \{..<\text{length } xs\}$
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* $\forall x \in \text{set } xs. (x, x) \in R$
assumes $(ys, n) \in \text{set-pmf } (\text{rquicksort } R \ xs)$
shows $\text{sorted-wrt } R \ ys \wedge \text{mset } ys = \text{mset } xs$
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 $(zs, n) \in \text{set-pmf } (\text{rquicksort } R \ ys)$ $\text{length } ys < \text{length } xs$ $\text{set } ys \subseteq \text{set } xs$ **for**
 $zs \ ys \ n$
using *that 1.IH total-on-subset[OF 1.prem(1) that(3)] 1.prem(2)* **by** *blast+*
show *?case*
proof (*cases xs = []*)
case *False*
with *1.prem* **obtain** $ls \ rs \ n1 \ n2 \ i$ **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 (*subst (asm) rquicksort.simps[of - xs]*) (*auto simp: Let-def o-def*)
note $ys = \langle ys = ls \ @ \ [xs \ ! \ i] \ @ \ rs \rangle$
define ls' **where** $ls' = [y \leftarrow \text{delete-index } i \ xs. (y, xs \ ! \ i) \in R]$
define rs' **where** $rs' = [y \leftarrow \text{delete-index } i \ xs. (y, xs \ ! \ i) \notin R]$
from $\langle i < \text{length } xs \rangle$ **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 $ls: (ls, n1) \in \text{set-pmf } (\text{rquicksort } R \ ls')$ **and** $rs: (rs, n2) \in \text{set-pmf } (\text{rquicksort } R \ rs')$
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] $\langle i < \text{length } xs \rangle$*
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 $x \in \text{set } ls$ for x*
proof –
from *that* **have** $x \in \# \text{mset } ls$ **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.prem 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.prem, 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 $\text{length } ys < \text{length } xs$ **mset** $ys \subseteq\# \text{mset } xs$ **for** ys
proof –
from $\text{set-mset-mono}[OF \text{ that}(2)]$ **have** $\text{set } ys \subseteq \text{set } xs$ **by** simp
also note $\langle \text{set } xs \subseteq A \rangle$
finally have $\text{set } ys \subseteq A$.
moreover from $\langle \text{distinct } xs \rangle$ **and** $\text{that}(2)$ **have** $\text{distinct } ys$
by $(\text{rule } \text{distinct-mset-mono})$
ultimately show $?thesis$ **using** that **and** $1.IH$ **by** blast
qed
define n **where** $n = \text{length } xs$
define cnt **where** $\text{cnt} = (\lambda i. \text{length } [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs ! i) \in R])$
have cnt-althdef : $\text{cnt } i = \text{linorder-rank } R (\text{set } xs) (xs ! i)$ **if** $i: i < n$ **for** i
proof –
have $\text{cnt } i = \text{length } [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs ! i) \in R]$ **by** $(\text{simp add: cnt-def})$
also have $\dots = \text{card } (\text{set } [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs ! i) \in R])$
by $(\text{intro } \text{distinct-card } [\text{symmetric}] \text{distinct-filter } \text{distinct-delete-index } 1.\text{prems})$
also have $\text{set } [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs ! i) \in R] =$
 $\{x \in \text{set } xs - \{xs ! i\}. (x, xs ! i) \in R\}$
using $1.\text{prems}$ **and** i **by** $(\text{simp add: set-delete-index-distinct } n\text{-def})$
also have $\text{card } \dots = \text{linorder-rank } R (\text{set } xs) (xs ! i)$ **by** $(\text{simp add: linorder-rank-def})$
finally show $?thesis$.
qed

from $1.\text{prems}$ **have** $\text{bij-betw } (!! \text{ } xs) \{.. < n\} (\text{set } xs)$
by $(\text{intro } \text{bij-betw-byWitness}[\text{where } f' = \text{index } xs]) (\text{auto simp: } n\text{-def } \text{index-nth-id})$
moreover have $\text{bij-betw } (\text{linorder-rank } R (\text{set } xs)) (\text{set } xs) \{.. < \text{card } (\text{set } xs)\}$
using $\text{assms}(1)$ **by** $(\text{rule } \text{bij-betw-linorder-rank}) (\text{insert } 1.\text{prems}, \text{auto})$
ultimately have $\text{bij-betw } (\text{linorder-rank } R (\text{set } xs) \circ (\lambda i. xs ! i)) \{.. < n\} \{.. < \text{card } (\text{set } xs)\}$
by $(\text{rule } \text{bij-betw-trans})$
hence bij : $\text{bij-betw } (\lambda i. \text{linorder-rank } R (\text{set } xs) (xs ! i)) \{.. < n\} \{.. < n\}$
using $1.\text{prems}$ **by** $(\text{simp add: } n\text{-def } o\text{-def } \text{distinct-card})$

show $?case$
proof $(\text{cases } xs = [])$
case False
hence $n > 0$ **by** $(\text{simp add: } n\text{-def})$
hence $[\text{simp}]: n \neq 0$ **by** $(\text{intro } \text{notI}) \text{auto}$
from False **have** $\text{map-pmf snd } (\text{rquicksort } R \text{ } xs) =$
 $\text{pmf-of-set } \{.. < \text{length } xs\} \gg=$
 $(\lambda i. \text{map-pmf } (\lambda z. \text{length } xs - 1 + \text{fst } z + \text{snd } z)$
 $(\text{pair-pmf } (\text{map-pmf snd } (\text{rquicksort } R [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs !$
 $i) \in R]))$
 $(\text{map-pmf snd } (\text{rquicksort } R [y \leftarrow \text{delete-index } i \text{ } xs. (y, xs !$
 $i) \in R])))$
by $(\text{subst } \text{rquicksort.simps})$
 $(\text{simp add: map-bind-pmf bind-map-pmf Let-def case-prod-unfold } o\text{-def } \text{pair-pmf-def})$

```

also have ... = pmf-of-set {.. $\text{length } xs$ }  $\gg$ 
  ( $\lambda i$ . map-pmf ( $\lambda z$ .  $n - 1 + \text{fst } z + \text{snd } z$ )
    (pair-pmf (rqs-cost (cnt i)) (rqs-cost (n - 1 - cnt i))))
proof (intro bind-pmf-cong refl, goal-cases)
  case (1 i)
  with  $\langle xs \neq [] \rangle$  have  $i : i < \text{length } xs$  by auto
  from i have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i)  $\notin$ 
R]) =
    rqs-cost (length [y←delete-index i xs. (y, xs ! i)  $\notin$  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)  $\notin$  R] = n - 1 - cnt i
  unfolding n-def cnt-def
  using sum-length-filter-compl[of  $\lambda y$ . (y, xs ! i)  $\in$  R delete-index i xs] i by
simp
  finally have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i)  $\notin$ 
R]) =
    rqs-cost (n - 1 - cnt i) .
  moreover have map-pmf snd (rquicksort R [y←delete-index i xs. (y, xs ! i)
 $\in$  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 {.. $n$ })  $\gg$ 
  ( $\lambda i$ . map-pmf ( $\lambda z$ .  $n - 1 + \text{fst } z + \text{snd } z$ ) (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 ( $\lambda i$ . linorder-rank R (set xs) (xs ! i)) (pmf-of-set {.. $n$ })
  using  $\langle n > 0 \rangle$  by (intro map-pmf-cong refl, subst (asm) set-pmf-of-set) (auto
simp: cnt-altdef)
also from  $\langle n > 0 \rangle$  have ... = pmf-of-set {.. $n$ } by (intro map-pmf-of-set-bij-betw
bij) auto
  also have pmf-of-set {.. $n$ }  $\gg$  ?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 $rqs\text{-}cost$, i.e. the expected number of comparisons of QuickSort when run on

a list with distinct elements.

fun *rqs-cost-exp* :: *nat* \Rightarrow *real* **where**

rqs-cost-exp 0 = 0

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

($\sum a \leq n. \text{inverse } (\text{real } (\text{Suc } n)) * \text{measure-pmf.expectation } (\text{rqs-cost } a \gg (\lambda a a. \text{rqs-cost } (n - a) \gg (\lambda b. \text{return-pmf } (n + a + b))))$) *real*)

unfolding *rqs-cost.simps* **by** (*subst pmf-expectation-bind-pmf-of-set*) *auto*

also have ... = ($\sum i \leq n. \text{inverse } (\text{real } (\text{Suc } n)) * (\text{real } n + \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i))$)

proof (*intro sum.cong refl, goal-cases*)

case (1 *i*)

have *rqs-cost* *i* $\gg (\lambda a. \text{rqs-cost } (n - i) \gg (\lambda b. \text{return-pmf } (n + a + b))) = \text{map-pmf } (\lambda(a,b). n + a + b) (\text{pair-pmf } (\text{rqs-cost } i) (\text{rqs-cost } (n - i)))$

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*))) ($\lambda z. \text{real } n + (\text{real } (\text{fst } z) + \text{real } (\text{snd } z))$)

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*)))

($\lambda z. \text{real } (\text{fst } z) + \text{real } (\text{snd } z)$) (**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 $\dots = (\sum_{i \leq n}. \text{inverse} (\text{real} (\text{Suc } n)) * \text{real } n) +$
 $(\sum_{i \leq n}. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i)) / \text{real} (\text{Suc } n)$
by (*simp add: sum.distrib field-simps sum-distrib-left sum-distrib-right*
sum-divide-distrib [symmetric] del: of-nat-Suc)
also have $(\sum_{i \leq n}. \text{inverse} (\text{real} (\text{Suc } n)) * \text{real } n) = \text{real } n$ **by** *simp*
also have $\dots + (\sum_{i \leq n}. \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n - i)) / \text{real} (\text{Suc } n) =$
 $\text{rqs-cost-exp } (\text{Suc } n)$
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'*:
 $\text{rqs-cost-exp } (\text{Suc } n) = \text{real } n + 2 / \text{real} (\text{Suc } n) * (\sum_{i \leq n}. \text{rqs-cost-exp } i)$
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 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 - λi. n - i λi. n - i]*) *auto*
also have $\dots + \dots = 2 * \dots$ **by** *simp*
also have $\dots / \text{real} (\text{Suc } n) = 2 / \text{real} (\text{Suc } n) * (\sum_{i \leq n}. \text{rqs-cost-exp } i)$ **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:

theorem *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** $m >$
 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 rqs-cost-exp-Suc'*) (*simp-all add: field-simps*)
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 rqs-cost-exp-Suc'*) (*simp-all add: field-simps*)
have $rqs\text{-}cost\text{-}exp (Suc (Suc n)) * real (Suc (Suc n)) - rqs\text{-}cost\text{-}exp (Suc n) * real (Suc n) =$
 $real ((n+1)*(n+2)) - real (n*(n+1)) + 2 * rqs\text{-}cost\text{-}exp (Suc n)$
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 $rqs\text{-}cost\text{-}exp (Suc (Suc n)) * real (n+2) = rqs\text{-}cost\text{-}exp (Suc n) * real (n+3) + real (2*(n+1))$
by (*simp add: algebra-simps*)
hence $rqs\text{-}cost\text{-}exp (Suc (Suc n)) / real (n+3) =$
 $rqs\text{-}cost\text{-}exp (Suc n) / real (n+2) + real (2*(n+1)) / (real (n+2)*real (n+3))$
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*: $F n = 2 * (\sum k=1..n. real (k - 1) / real (k * (k + 1)))$ **for** n
proof (*cases n ≥ 1*)
case *True*
thus *?thesis* **by** (*induction n rule: dec-induct*) (*simp-all add: F-Suc 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 F-eq*)
also have $?S = (\sum k=1..n. 2 / real (Suc k) - 1 / real k)$
by (*intro sum.cong*) (*simp-all add: field-simps of-nat-diff*)
also have $\dots = 2 * (\sum k=1..n. inverse (real (Suc k))) - harm n$
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 - λx. x - 1 Suc]*) *auto*
also have $\dots = harm (Suc n) - 1$ **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 $\dots * real (n + 1) = 2 * real (n + 1) * harm n - 4 * real n$
by (*simp add: field-simps*)
also have $F n * real (n + 1) = rqs\text{-}cost\text{-}exp n$ **by** (*simp add: F-def add-ac*)
finally show *?thesis* .
qed

lemma *asympt-equiv-harm* [*asympt-equiv-intros*]: $harm \sim[at\text{-}top] (\lambda n. ln (real n))$
proof -
have $(\lambda n. harm n - ln (real n)) \in O(\lambda. 1)$ **using** *euler-mascheroni-LIMSEQ*
by (*intro bigoI-tendsto[where c = euler-mascheroni]*) *simp-all*
also have $(\lambda. 1) \in o(\lambda n. ln (real n))$ **by** *auto*

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*: $\text{rqs-cost-exp} \sim[\text{at-top}] (\lambda n. 2 * n * \ln n)$

proof –

have $\text{rqs-cost-exp} = (\lambda n. 2 * \text{real } (n + 1) * \text{harm } n - 4 * \text{real } n)$ **using** *rqs-cost-exp-eq ..*

also have $\dots = (\lambda n. 2 * \text{real } n * \text{harm } n + (2 * \text{harm } n - 4 * \text{real } n))$

by *(simp add: algebra-simps)*

finally have $\text{rqs-cost-exp} \sim[\text{at-top}] \dots$ **by** *simp*

also have $\dots \sim[\text{at-top}] (\lambda n. 2 * \text{real } n * \text{harm } n)$

proof *(subst asymp-equiv-add-right)*

have $(\lambda x. 1 * \text{harm } x) \in o(\lambda x. \text{real } x * \text{harm } x)$

by *(intro landau-o.small-big-mult smalllo-real-nat-transfer) simp-all*

moreover have $\text{harm} \in \omega(\lambda -. 1 :: \text{real})$

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-smalllo)*

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-smalllo) simp-all*

qed *simp-all*

also have $\dots \sim[\text{at-top}] (\lambda n. 2 * \text{real } n * \ln (\text{real } n))$ **by** *(intro asymp-equiv-intros)*

finally show *?thesis* .

qed

lemma *harm-mono*: $m \leq n \implies \text{harm } m \leq (\text{harm } n :: \text{real})$

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*: $n > 0 \implies \text{harm } n \geq (1 :: \text{real})$

using *harm-mono[of 1 n]* **by** *simp*

lemma *mono-rqs-cost-exp*: *mono rqs-cost-exp*

proof *(rule incseq-SucI)*

fix n **show** $\text{rqs-cost-exp } n \leq \text{rqs-cost-exp } (\text{Suc } n)$

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 $\dots \leq (\text{harm } n * 2 * (\text{real } n + 1) - 2 * \text{real } n) / (\text{real } n + 1)$ **using**

False

by *(intro divide-right-mono diff-right-mono mult-right-mono) (auto simp: harm-ge-1)*

also have $\dots = \text{rqs-cost-exp } (\text{Suc } n) - \text{rqs-cost-exp } n$

by *(simp add: rqs-cost-exp-eq harm-Suc field-simps)*

```

    finally show ?thesis by simp
qed auto
qed

```

```

lemma rqs-cost-exp-leI:  $m \leq n \implies \text{rqs-cost-exp } m \leq \text{rqs-cost-exp } n$ 
  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 ( $\lambda y. (x,y) \in R \wedge (y,x) \in R$ )  $xs$ ,
   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'* :: $('a \times 'a)$ set $\Rightarrow 'a$ list $\Rightarrow ('a$ list \times nat) pmf **where**

```

rquicksort'  $R xs =$ 
  (if  $xs = []$  then
    return-pmf ( $[], 0$ )
  else
    do {
       $i \leftarrow \text{pmf-of-set } \{..<\text{length } xs\}$ ;
      let  $x = xs ! i$ ;
      case threeway-partition  $x R$  (delete-index  $i xs$ ) of
        ( $ls, es, rs$ )  $\Rightarrow$  do {
          ( $ls, n1$ )  $\leftarrow$  rquicksort'  $R ls$ ;
          ( $rs, n2$ )  $\leftarrow$  rquicksort'  $R rs$ ;
          return-pmf ( $ls @ x \# es @ rs, \text{length } xs - 1 + n1 + n2$ )
        }
    })

```

by *auto*

termination proof (relation *Wellfounded.measure* (*length* \circ *snd*), *goal-cases*)

show *wf* (*Wellfounded.measure* (*length* \circ *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 *less* :: $('a \times 'a)$ set $\Rightarrow 'a \Rightarrow 'a$ list $\Rightarrow 'a$ list **where**

less R x 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 x xs = filter (λy. (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 *greater-Cons*:

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

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

qualified lemma *length-greater-less* [*intro*]:

assumes x ∈ set xs
shows *length* (*greater* R x xs) < *length* xs
using *assms* **by** (*induction xs*) (*auto simp: greater-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 (*greater* 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 (greater 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 $f :: 'a \Rightarrow 'b :: \{\text{banach, second-countable-topology}\}$
shows $\text{measure-pmf.expectation (pair-pmf p q) } (\lambda x. f (\text{fst } x)) = \text{measure-pmf.expectation } p f$
proof –
have $\text{measure-pmf.expectation (pair-pmf p q) } (\lambda x. f (\text{fst } x)) =$
 $\text{measure-pmf.expectation (map-pmf fst (pair-pmf p q)) } f$ **by** *simp*
also have $\text{map-pmf fst (pair-pmf p q) } = p$
by (*simp add: map-fst-pair-pmf*)
finally show ?thesis .
qed

lemma *expectation-pair-pmf-snd* [simp]:
fixes $f :: 'a \Rightarrow 'b :: \{\text{banach, second-countable-topology}\}$
shows $\text{measure-pmf.expectation (pair-pmf p q) } (\lambda x. f (\text{snd } x)) = \text{measure-pmf.expectation } q f$
proof –
have $\text{measure-pmf.expectation (pair-pmf p q) } (\lambda x. f (\text{snd } x)) =$
 $\text{measure-pmf.expectation (map-pmf snd (pair-pmf p q)) } f$ **by** *simp*
also have $\text{map-pmf snd (pair-pmf p q) } = q$
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 $\text{length (lesss R (xs ! i) xs) } \leq i$
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-greater R le-sorted*:
assumes *sorted-wrt R xs i < length xs*
shows $\text{length (greater R (xs ! i) xs) } \leq \text{length xs} - i - 1$
using *assms*

by (induction arbitrary: i rule: sorted-wrt.induct)
(force simp: greater-def nth-Cons le-Suc-eq split: nat.splits)+

qualified lemma *length-less-le'*:

assumes $i < \text{length } xs$ *linorder-on* A R set $xs \subseteq A$

shows $\text{length } (\text{less } R (\text{insort-wrt } R \text{ } xs \ ! \ i) \ xs) \leq i$

proof –

define x **where** $x = \text{insort-wrt } R \text{ } xs \ ! \ i$

define *less* **where** $\text{less} = (\lambda x \ y. (x,y) \in R \wedge (y,x) \notin R)$

have $\text{length } (\text{less } R \ x \ xs) = \text{size } \{\# \ y \in \# \ \text{mset } xs. \ \text{less } y \ x \ \#\}$

by (simp add: less-def size-mset [symmetric] less-def mset-filter del: size-mset)

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

also have $\text{size } \{\# \ y \in \# \ \text{mset } (\text{insort-wrt } R \text{ } xs). \ \text{less } y \ x \ \#\} =$
 $\text{length } (\text{less } R \ x \ (\text{insort-wrt } R \text{ } xs))$

by (simp only: mset-filter [symmetric] size-mset less-def less-def)

also have $\dots \leq i$ **unfolding** x -def **by** (rule *length-less-le-sorted*) (use *assms* in *auto*)

finally show *?thesis* **unfolding** x -def .

qed

qualified lemma *length-greater-le'*:

assumes $i < \text{length } xs$ *linorder-on* A R set $xs \subseteq A$

shows $\text{length } (\text{greater } R (\text{insort-wrt } R \text{ } xs \ ! \ i) \ xs) \leq \text{length } xs - i - 1$

proof –

define x **where** $x = \text{insort-wrt } R \text{ } xs \ ! \ i$

define *less* **where** $\text{less} = (\lambda x \ y. (x,y) \in R \wedge (y,x) \notin R)$

have $\text{length } (\text{greater } R \ x \ xs) = \text{size } \{\# \ y \in \# \ \text{mset } xs. \ \text{less } x \ y \ \#\}$

by (simp add: greater-def size-mset [symmetric] less-def mset-filter del: size-mset)

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

also have $\text{size } \{\# \ y \in \# \ \text{mset } (\text{insort-wrt } R \text{ } xs). \ \text{less } x \ y \ \#\} =$
 $\text{length } (\text{greater } R \ x \ (\text{insort-wrt } R \text{ } xs))$

by (simp only: mset-filter [symmetric] size-mset greater-def less-def)

also have $\dots \leq \text{length } (\text{insort-wrt } R \text{ } xs) - i - 1$ **unfolding** x -def

by (rule *length-greater-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 $\text{measure-pmf.expectation } (\text{rqs}'\text{-cost } R \text{ } xs) \text{ real} \leq \text{rqs-cost-exp } (\text{length } xs)$

using *assms*

proof (induction $R \text{ } xs$ rule: *rqs'-cost.induct*)

case (1 $R \text{ } xs$)

show *?case*

proof (cases $xs = []$)

case *False*

```

define  $n$  where  $n = \text{length } xs - 1$ 
have  $\text{length-eq} : \text{length } xs = \text{Suc } n$  using  $\text{False}$  by ( $\text{simp add: } n\text{-def}$ )
define  $E$  where  $E = (\lambda xs. \text{measure-pmf.expectation } (rqs'\text{-cost } R \text{ } xs) \text{ real})$ 
define  $f$  where  $f = (\lambda x. rqs\text{-cost-exp } (\text{length } (\text{less } R \text{ } x \text{ } xs)) +$ 
 $\quad rqs\text{-cost-exp } (\text{length } (\text{greater } R \text{ } x \text{ } xs)))$ 
have  $rqs'\text{-cost } R \text{ } xs =$ 
  do {
     $i \leftarrow \text{pmf-of-set } \{..<\text{length } xs\};$ 
     $\text{map-pmf } (\lambda(n1, y). \text{length } xs - \text{Suc } 0 + n1 + y)$ 
    ( $\text{pair-pmf } (rqs'\text{-cost } R \text{ } (\text{less } R \text{ } (xs ! i) \text{ } xs))$ 
      ( $rqs'\text{-cost } R \text{ } (\text{greater } R \text{ } (xs ! i) \text{ } xs)))$ )
  }
using  $\text{False}$  by ( $\text{subst } rqs'\text{-cost.simps}$ ) ( $\text{simp-all add: } \text{Let-def}$ )
also have  $\text{measure-pmf.expectation } \dots \text{ real} = \text{real } n +$ 
 $(\sum k < \text{length } xs. E \text{ } (\text{less } R \text{ } (xs ! k) \text{ } xs) + E \text{ } (\text{greater } R \text{ } (xs ! k) \text{ } xs)) /$ 
 $\text{real } (\text{length } xs)$ 
using  $\text{False}$ 
by ( $\text{subst pmf-expectation-bind-pmf-of-set}$ )
  ( $\text{auto intro!: finite-imageI finite-cartesian-product simp: case-prod-unfold}$ 
     $\text{integrable-measure-pmf-finite sum-divide-distrib [symmetric] field-simps}$ 
     $\text{length-eq sum.distrib } E\text{-def}$ )
also have  $\dots \leq \text{real } n + (\sum k < \text{length } xs. f \text{ } (xs ! k)) / \text{real } (\text{length } xs)$ 
unfolding  $E\text{-def } f\text{-def}$  using  $\text{False } 1.\text{prems}$ 
by ( $\text{intro add-mono order.refl divide-right-mono sum-mono } 1.\text{IH}[OF \text{ } - \text{ refl}]$ 
 $\text{False}$ )
  ( $\text{auto simp: less-def greater-def}$ )
also have  $(\sum k < \text{length } xs. f \text{ } (xs ! k)) = (\sum x \in \#mset \text{ } xs. f \text{ } x)$ 
by ( $\text{simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth}$ )
  ( $\text{simp-all add: atLeast0LessThan}$ )
also have  $mset \text{ } xs = mset \text{ } (\text{insort-wrt } R \text{ } xs)$ 
by  $\text{simp}$ 
also have  $(\sum x \in \#\dots f \text{ } x) = (\sum i < \text{length } xs. f \text{ } (\text{insort-wrt } R \text{ } xs ! i))$ 
by ( $\text{simp only: mset-map [symmetric] sum-mset-sum-list sum-list-sum-nth}$ )
  ( $\text{simp-all add: atLeast0LessThan}$ )
also have  $\dots \leq (\sum i < \text{length } xs. rqs\text{-cost-exp } i + rqs\text{-cost-exp } (\text{length } xs - i -$ 
 $1))$ 
unfolding  $f\text{-def}$ 
proof ( $\text{intro sum-mono add-mono } rqs\text{-cost-exp-leI}$ )
fix  $i$  assume  $i \in \{..<\text{length } xs\}$ 
show  $\text{length } (\text{less } R \text{ } (\text{insort-wrt } R \text{ } xs ! i) \text{ } xs) \leq i$ 
using  $i 1.\text{prems}$  by ( $\text{intro length-less-le' [where } A = A]$ )  $\text{auto}$ 
show  $\text{length } (\text{greater } R \text{ } (\text{insort-wrt } R \text{ } xs ! i) \text{ } xs) \leq \text{length } xs - i - 1$ 
using  $i 1.\text{prems}$  by ( $\text{intro length-greater-le' [where } A = A]$ )  $\text{auto}$ 
qed
also have  $\dots = (\sum i \leq n. rqs\text{-cost-exp } i + rqs\text{-cost-exp } (n - i))$ 
by ( $\text{intro sum.cong}$ ) ( $\text{auto simp: length-eq}$ )
also have  $\text{real } n + \dots / \text{real } (\text{length } xs) = rqs\text{-cost-exp } (\text{length } xs)$ 
by ( $\text{simp add: length-eq } rqs\text{-cost-exp.simps}(2)$ )
finally show  $?thesis$  by ( $\text{simp add: divide-right-mono}$ )

```

```

  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 (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-quick-sort [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.prem1 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-quick-sort'*:

assumes *linorder-on A R* **and** *set xs* \subseteq *A*

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

by (*rule sorted-wrt-quick-sort'*)

(*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-quick-sort'* [*simp*]: *fst (quicksort' R xs) = quicksort R xs*

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

lemma *snd-quick-sort'* [*simp*]: *snd (quicksort' R xs) = qs-cost R xs*

by (*induction R xs rule: quick-sort.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 psub-
set.premss)
  also from A have card A > 0 by (intro Nat.gr0I) auto
  hence {.. $\text{card } A$ } = {.. $n$ } by (auto simp: n-def)
  also have map-pmf ( $\lambda m. n + m$ ) (
    do {
      i  $\leftarrow$  pmf-of-set {.. $n$ };
      (n1, n2)  $\leftarrow$  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 $\text{rqs-cost-exp } n = 2 * \text{real } (n + 1) * \text{harm } n - 4 * \text{real } n$ and $\text{rqs-cost-exp} \sim[\text{sequentially}] (\lambda x. 2 * \text{real } x * \ln (\text{real } x))$ before.

corollary *expectation-qs-cost:*

assumes *finite A and linorder-on B R and $A \subseteq B$*

defines *random-list \equiv 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.