

# 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

<b>1</b>	<b>Randomised QuickSort</b>	<b>2</b>
1.1	Deletion by index . . . . .	2
1.2	Definition . . . . .	3
1.3	Correctness proof . . . . .	3
1.4	Cost analysis . . . . .	4
1.5	Expected cost . . . . .	4
1.6	Version for lists with repeated elements . . . . .	5
<b>2</b>	<b>Average case analysis of deterministic QuickSort</b>	<b>8</b>
2.1	Definition of deterministic QuickSort . . . . .	8
2.2	Analysis . . . . .	9

# 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
  <proof>
```

```
lemma delete-index-ge-length:  $n \geq \text{length } xs \implies \text{delete-index } n \text{ } xs = xs$ 
  <proof>
```

```
lemma length-delete-index [simp]:  $n < \text{length } xs \implies \text{length } (\text{delete-index } n \text{ } xs) =$ 
 $\text{length } xs - 1$ 
  <proof>
```

```
lemma delete-index-Cons:
  delete-index n (x # xs) = (if n = 0 then xs else x # delete-index (n - 1) xs)
  <proof>
```

```
lemma insert-set-delete-index:
   $n < \text{length } xs \implies \text{insert } (xs ! n) (\text{set } (\text{delete-index } n \text{ } xs)) = \text{set } xs$ 
  <proof>
```

```
lemma add-mset-delete-index:
   $i < \text{length } xs \implies \text{add-mset } (xs ! i) (\text{mset } (\text{delete-index } i \text{ } xs)) = \text{mset } xs$ 
  <proof>
```

```
lemma nth-delete-index:
   $i < \text{length } xs \implies n < \text{length } xs \implies$ 
   $\text{delete-index } n \text{ } xs ! i = (\text{if } i < n \text{ then } xs ! i \text{ else } xs ! \text{Suc } i)$ 
  <proof>
```

```
lemma set-delete-index-distinct:
  assumes distinct xs  $n < \text{length } xs$ 
  shows  $\text{set } (\text{delete-index } n \text{ } xs) = \text{set } xs - \{xs ! n\}$ 
  <proof>
```

**lemma** *distinct-delete-index* [simp, intro]:

**assumes** *distinct xs*

**shows** *distinct (delete-index n xs)*

*<proof>*

**lemma** *mset-delete-index* [simp]:

$i < \text{length } xs \implies \text{mset } (\text{delete-index } i \text{ } xs) = \text{mset } xs - \{\# \text{ } xs!i \#\}$

*<proof>*

## 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)*

*}*

*})*

*<proof>*

**termination** *<proof>*

**declare** *rquicksort.simps* [simp del]

**lemma** *rquicksort-Nil* [simp]: *rquicksort R [] = return-pmf ([], 0)*

*<proof>*

## 1.3 Correctness proof

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

$xs \neq [] \implies \text{set-pmf } (\text{pmf-of-set } \{..*length xs*\}) = \{..*length xs*\}$

*<proof>*

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

**shows**  $\text{sorted-wrt } R \text{ } ys \wedge \text{mset } ys = \text{mset } xs$

*<proof>*

## 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* ::  $\text{nat} \Rightarrow \text{nat pmf}$  **where**

*rqs-cost* 0 = *return-pmf* 0

| *rqs-cost* (Suc n) =

*do* {  $i \leftarrow \text{pmf-of-set } \{..n\}$ ;  $a \leftarrow \text{rqs-cost } i$ ;  $b \leftarrow \text{rqs-cost } (n - i)$ ; *return-pmf* (n + a + b)}

**lemma** *finite-set-pmf-rqs-cost* [*intro!*]:  $\text{finite } (\text{set-pmf } (\text{rqs-cost } n))$

*<proof>*

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  $\subseteq$  A and distinct xs*

**shows**  $\text{map-pmf } \text{snd } (\text{rquicksort } R \text{ } xs) = \text{rqs-cost } (\text{length } xs)$

*<proof>*

## 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* ::  $\text{nat} \Rightarrow \text{real}$  **where**

*rqs-cost-exp* 0 = 0

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

**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*]:  $\text{rqs-cost-exp } (\text{Suc } 0) = 0$  *<proof>*

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*⟩

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'*:  
 $rqs\text{-}cost\text{-}exp\ (Suc\ n) = real\ n + 2 / real\ (Suc\ n) * (\sum\ i \leq n. rqs\text{-}cost\text{-}exp\ i)$   
 ⟨*proof*⟩

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*]:  $rqs\text{-}cost\text{-}exp\ n = 2 * real\ (n + 1) * harm\ n - 4 * real\ n$   
 ⟨*proof*⟩

**lemma** *asympt-equiv-harm* [*asympt-equiv-intros*]:  $harm \sim[at\text{-}top] (\lambda n. ln\ (real\ n))$   
 ⟨*proof*⟩

**corollary** *rqs-cost-exp-asympt-equiv*:  $rqs\text{-}cost\text{-}exp \sim[at\text{-}top] (\lambda n. 2 * n * ln\ n)$   
 ⟨*proof*⟩

**lemma** *harm-mono*:  $m \leq n \implies harm\ m \leq (harm\ n :: real)$   
 ⟨*proof*⟩

**lemma** *harm-Suc-0* [*simp*]:  $harm\ (Suc\ 0) = 1$   
 ⟨*proof*⟩

**lemma** *harm-ge-1*:  $n > 0 \implies harm\ n \geq (1 :: real)$   
 ⟨*proof*⟩

**lemma** *mono-rqs-cost-exp*: *mono rqs-cost-exp*  
 ⟨*proof*⟩

**lemma** *rqs-cost-exp-leI*:  $m \leq n \implies rqs\text{-}cost\text{-}exp\ m \leq rqs\text{-}cost\text{-}exp\ n$   
 ⟨*proof*⟩

## 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 × '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)
        }
    })
  ⟨proof⟩

```

**termination** ⟨proof⟩

**declare** *rquicksort'.simps* [*simp del*]

**lemma** *rquicksort'-Nil* [*simp*]: *rquicksort'* R [] = return-pmf ([], 0)  
 ⟨proof⟩

**context**  
**begin**

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

```

less R x (y # ys) =
  (if (y, x) ∈ R ∧ (x, y) ∉ R then y # less R x ys else less R x ys)
  ⟨proof⟩ lemma length-less-le [intro]: length (less R x xs) ≤ length xs
  ⟨proof⟩ lemma length-less-less [intro]:

```

**assumes** x ∈ set xs

**shows** *length* (*less* R x xs) < *length* xs

⟨proof⟩ **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)
  ⟨proof⟩ lemma length-greater-le [intro]: length (greater R x xs) ≤ length xs
  ⟨proof⟩ lemma length-greater-less [intro]:

```

**assumes** x ∈ set xs

**shows** *length* (*greater* R x xs) < *length* xs

*<proof>*

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)))
    })
```

*<proof>*

**termination** *<proof>*

**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)
  }
```

*<proof>*

**lemma** *finite-set-pmf-rqs'-cost* [*simp, intro*]:

```
finite (set-pmf (rqs'-cost R xs))
<proof>
```

**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>
```

**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>* **lemma** *length-lesss-le-sorted*:

```
assumes sorted-wrt R xs i < length xs
shows length (lesss R (xs ! i) xs) ≤ i
```

```

⟨proof⟩ lemma length-greaters-le-sorted:
  assumes sorted-wrt R xs i < length xs
  shows length (greaters R (xs ! i) xs) ≤ length xs - i - 1
⟨proof⟩ lemma length-less-le':
  assumes i < length xs linorder-on A R set xs ⊆ A
  shows length (less R (insort-wrt R xs ! i) xs) ≤ i
⟨proof⟩ 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⟩

```

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)
⟨proof⟩

```

```

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
⟨proof⟩

```

```

corollary set-quicksort [simp]: set (quicksort R xs) = set xs
⟨proof⟩

```

**theorem** *sorted-wrt-quicksort*:

**assumes** *trans*  $R$  **and** *total-on* (*set*  $xs$ )  $R$  **and**  $\bigwedge x. x \in \text{set } xs \implies (x, x) \in R$

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

*<proof>*

**corollary** *sorted-wrt-quicksort'*:

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

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

*<proof>*

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* ( $\lambda y. (y, x) \in R$ )  $xs$ ;

  ( $ls', n1$ ) = *quicksort'*  $R$   $ls$ ;

  ( $rs', n2$ ) = *quicksort'*  $R$   $rs$

*in*

  ( $ls' @ [x] @ rs', \text{length } xs + n1 + n2$ ))

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*  $R$  ( $x \# xs$ ) =

$\text{length } xs + \text{qs-cost } R (\text{filter } (\lambda y. (y, x) \in R) xs) + \text{qs-cost } R (\text{filter } (\lambda y. (y, x) \notin R) xs)$

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*  $R$   $xs$

*<proof>*

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

*<proof>*

## 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**  $\text{map-pmf } (qs\text{-cost } R) (\text{pmf-of-set } (\text{permutations-of-set } A)) = rqs\text{-cost}$   
 $(\text{card } A)$   
 ⟨*proof*⟩

We therefore have the same expectation as well. (Note that we showed  $rqs\text{-cost-exp } n = 2 * \text{real } (n + 1) * \text{harm } n - 4 * \text{real } n$  and  $rqs\text{-cost-exp} \sim[\text{sequentially}] (\lambda x. 2 * \text{real } x * \ln (\text{real } x))$  before.

**corollary** *expectation-qs-cost:*

**assumes** *finite A and linorder-on B R and  $A \subseteq B$*   
**defines**  $\text{random-list} \equiv \text{pmf-of-set } (\text{permutations-of-set } A)$   
**shows**  $\text{measure-pmf.expectation } (\text{map-pmf } (qs\text{-cost } R) \text{ random-list}) \text{ real} =$   
 $rqs\text{-cost-exp } (\text{card } A)$   
 ⟨*proof*⟩

**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.