Cost Analysis of QuickSort

Manuel Eberl

August 16, 2018

Abstract

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

Contents

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

2 Average case analysis of deterministic QuickSort \hfill 8
  2.1 Definition of deterministic QuickSort \hfill 8
  2.2 Analysis \hfill 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 ⇒ '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
  ⟨proof⟩

lemma delete-index-ge-length: \( n \geq \) length xs ⇒ delete-index n xs = xs
  ⟨proof⟩

lemma length-delete-index [simp]: \( n < \) length xs ⇒ length (delete-index n xs) = 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 < length xs ⇒ insert (xs ! n) (set (delete-index n xs)) = set xs
  ⟨proof⟩

lemma add-mset-delete-index:
  i < length xs ⇒ add-mset (xs ! i) (mset (delete-index i xs)) = mset xs
  ⟨proof⟩

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)
  ⟨proof⟩

lemma set-delete-index-distinct:
  assumes distinct xs n < length xs
  shows set (delete-index n xs) = 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 < length xs ⇒ mset (delete-index i xs) = mset xs − {# 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 ≠ [] ⇒ set-pmf (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 quicksort-correct:**

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

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

*Shows* \( \text{sorted-wrt } R \ ys \land \text{mset } ys = \text{mset } xs \)

\((\text{proof})\)

### 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 :: nat ⇒ nat pmf** where

\[ rqs\text{-cost } 0 = \text{return-pmf } 0 \]

\[ rqs\text{-cost } (\text{Suc } n) = \]

\[ \begin{array}{l}
\text{do } \{ i \leftarrow \text{pmf-of-set } \{..n\}; a \leftarrow rqs\text{-cost } i; b \leftarrow rqs\text{-cost } (n - i); \text{return-pmf } (n + a + b) \}
\end{array} \]

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

\((\text{proof})\)

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

**Theorem snd-rquicksort:**

*Assumes* \( \text{linorder-on } A \ R \) and \( \text{set } xs \subseteq A \) and \( \text{distinct } xs \)

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

\((\text{proof})\)

### 1.5 Expected cost

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

**fun rqs-cost-exp :: nat ⇒ real** where

\[ rqs\text{-cost-exp } 0 = 0 \]

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

**Lemmas**

\[ rqs\text{-cost-exp-0 } = rqs\text{-cost-exp-simps}(1) \]

**Lemmas**

\[ rqs\text{-cost-exp-Suc } [\text{simp del}] = rqs\text{-cost-exp-simps}(2) \]

**Lemma**

\[ rqs\text{-cost-exp-Suc-0 } [\text{simp}]: rqs\text{-cost-exp } (\text{Suc } 0) = 0 \]

\((\text{proof})\)
The following theorem shows that \( \text{rqs-cost-exp} \) is indeed the expectation of \( \text{rqs-cost} \).

**Theorem** \( \text{expectation-rqs-cost} \): \( \text{measure-pmf.expectation (rqs-cost n)} \) \( \text{real} = \text{rqs-cost-exp n} \) (proof)

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

**Lemma** \( \text{rqs-cost-exp-Suc'} \):
\[ \text{rqs-cost-exp (Suc n)} = \text{real n} + 2 / \text{real (Suc n)} * (\sum_{i \leq n} \text{rqs-cost-exp i}) \] (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** \( \text{rqs-cost-exp-eq} \): \( \text{rqs-cost-exp n} = 2 * \text{real (n + 1)} * \text{harm n} - 4 * \text{real n} \) (proof)

**Lemma** \( \text{asymp-equiv-harm} \): \( \text{harm} \sim \lambda n. \ln (\text{real n}) \) (proof)

**Corollary** \( \text{rqs-cost-exp-asym-equiv} \): \( \text{rqs-cost-exp} \sim \lambda n. 2 * n * \ln n \) (proof)

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

**Lemma** \( \text{harm-Suc-0} \): \( \text{harm (Suc 0)} = 1 \) (proof)

**Lemma** \( \text{harm-ge-1} \): \( n > 0 \implies \text{harm n} \geq (1 :: \text{real}) \) (proof)

**Lemma** \( \text{mono-rqs-cost-exp} \): mono \( \text{rqs-cost-exp} \) (proof)

**Lemma** \( \text{rqs-cost-exp-leI} \): \( m \leq n \implies \text{rqs-cost-exp m} \leq \text{rqs-cost-exp n} \) (proof)

### 1.6 Version for lists with repeated elements

**Definition** \( \text{threeway-partition} \) where

\[
\text{threeway-partition x R xs} = \\
(\text{filter (\lambda y. (y,x) \in R \land (x,y) \notin R)} \text{ xs}, \\
(\text{filter (\lambda y. (x,y) \in R \land (y,x) \in R)} \text{ xs}, \\
(\text{filter (\lambda y. (x,y) \in R \land (y,x) \notin R)} \text{ xs})
\]
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.

\[
\text{function } \text{rquicksort} :: ('a \times 'a) \text{ set } \Rightarrow 'a \text{ list } \Rightarrow ('a \text{ list } \times \text{ nat}) \text{ pmf where}
\]

\[
\text{rquicksort} \ R \ xs =
\]

\[
\begin{cases}
\text{return-pmf } ([], 0) & \text{if } xs = [] \\
\text{do }
\]

\[
i \leftarrow \text{pmf-of-set } \{ ..< \text{length } xs \};
\]

\[
\text{let } x = xs ! i;
\]

\[
\text{case threeway-partition } x \ R \ (\text{delete-index } i \ xs) \text{ of } (ls, es, rs) \Rightarrow \text{do }
\]

\[
\begin{cases}
(ls, n1) \leftarrow \text{rquicksort} \ R \ ls; \\
(rs, n2) \leftarrow \text{rquicksort} \ R \ rs;
\end{cases}
\]

\[
\text{return-pmf } (\langle \text{ls @ } x @ \text{es @ } rs, \text{length } xs - 1 + n1 + n2 \rangle)
\]

\end{cases}
\]

\[
\text{⟨proof⟩} \]

\text{termination} \ ⟨proof⟩

\text{declare} \ \text{rquicksort}.\text{simps} [\text{simp del}]

\text{lemma} \ \text{rquicksort}'-\text{Nil} [\text{simp}; \text{rquicksort}' \ R \ [] = \text{return-pmf } ([], 0) \ ⟨proof⟩]

\text{context}

\text{begin}

\text{qualified definition} \ \text{lesss} :: ('a \times 'a) \text{ set } \Rightarrow 'a \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list where}

\[
\text{lesss} \ R \ x \ xs = \text{filter } (\lambda y. (y, x) \in R \land (x, y) \notin R) \ xs
\]

\text{qualified definition} \ \text{greaters} :: ('a \times 'a) \text{ set } \Rightarrow 'a \Rightarrow 'a \text{ list } \Rightarrow 'a \text{ list where}

\[
\text{greaters} \ R \ x \ xs = \text{filter } (\lambda y. (x, y) \in R \land (y, x) \notin R) \ xs
\]

\text{qualified lemma} \ \text{lesss-Cons}:

\[
\text{lesss} \ R \ x \ (y \# \ ys) =
\]

\[
\begin{cases}
\text{if } (y, x) \in R \land (x, y) \notin R \text{ then } y \# \ \text{lesss} \ R \ x \ ys \ \text{else } \text{lesss} \ R \ x \ ys
\end{cases}
\]

\langle proof \rangle \ \text{lemma} \ \text{length-lesss-le} [\text{intro}]: \text{length } (\text{lesss } R \ x \ xs) \leq \text{length } xs

\langle proof \rangle \ \text{lemma} \ \text{length-lesss-less} [\text{intro}]:

\text{assumes } x \in \text{set } xs

\text{shows } \text{length } (\text{lesss } R \ x \ xs) < \text{length } xs

\langle proof \rangle \ \text{lemma} \ \text{greaters-Cons}:

\[
\text{greaters} \ R \ x \ (y \# \ ys) =
\]

\[
\begin{cases}
\text{if } (x, y) \in R \land (y, x) \notin R \text{ then } y \# \ \text{greaters} \ R \ x \ ys \ \text{else } \text{greaters} \ R \ x \ ys
\end{cases}
\]

\langle proof \rangle \ \text{lemma} \ \text{length-greaters-le} [\text{intro}]: \text{length } (\text{greaters } R \ x \ xs) \leq \text{length } xs

\langle proof \rangle \ \text{lemma} \ \text{length-greaters-less} [\text{intro}]:

\text{assumes } x \in \text{set } xs

\text{shows } \text{length } (\text{greaters } R \ x \ xs) < \text{length } xs

\text{6}
The following function counts the comparisons made by the modified randomised Quicksort.

```plaintext
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)))
}
⟨proof⟩
termination ⟨proof⟩
```

```plaintext
declare rqs'-cost.simps [simp del]
```

```plaintext
lemma rqs'-cost-nonempty:
xzs ≠ [] ⇒ 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)
}
⟨proof⟩
```

```plaintext
lemma finite-set-pmf-rqs'-cost [simp, intro]:
finite (set-pmf (rqs'-cost R xs))
⟨proof⟩
```

```plaintext
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⟩
```

```plaintext
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
```

\begin{proof}
\textbf{lemma length-greaters-le-sorted:}
\textbf{assumes} sorted-wrt R xs i < length xs
\textbf{shows} \quad length (greaters R (xs ! i) xs) \leq length xs - i - 1
\end{proof}

\begin{proof}
\textbf{lemma length-lesss-le′:}
\textbf{assumes} i < length xs linorder-on A R set xs \subseteq A
\textbf{shows} \quad length (lesss R (insort-wrt R xs ! i) xs) \leq i
\end{proof}

\begin{proof}
\textbf{lemma length-greaters-le′:}
\textbf{assumes} i < length xs linorder-on A R set xs \subseteq A
\textbf{shows} \quad length (greaters R (insort-wrt R xs ! i) xs) \leq length xs - i - 1
\end{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.

\begin{theorem}
\textbf{rqs′-cost-expectation-le:}
\textbf{assumes} linorder-on A R set xs \subseteq A
\textbf{shows} \quad measure-pmf.expectation (rqs′-cost R xs) real \leq rqs-cost-exp (length xs)
\end{theorem}

\end{proof}

\section{Average case analysis of deterministic QuickSort}

\textit{theory} Quick-Sort-Average-Case
\textit{imports} Randomised-Quick-Sort
\begin{proof}
\subsection{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 \cite{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.

\textbf{fun} quicksort :: \(\langle\text{‘a \times ‘a}\rangle\) set \(\Rightarrow\) ‘a list \(\Rightarrow\) ‘a list \textbf{where}
\quad quicksort - [] = []
\quad quicksort R (x \# xs) =
\quad \quad quicksort R (filter (\lambda y. (y, x) \in R) xs) @ [x] @ quicksort R (filter (\lambda y. (y, x) \notin R) xs)

We can easily show that this QuickSort is correct:

\begin{theorem}
\textbf{mset-quicksort} [simp]: mset (quicksort R xs) = mset xs
\end{theorem}

\begin{corollary}
\textbf{set-quicksort} [simp]: set (quicksort R xs) = set xs
\end{corollary}

\end{proof}
\begin{proof}
\end{proof}

\textbf{Theorem sorted-wrt-quicksort:\hspace{0.5cm}}
\textbf{Assumes} trans \( R \) \textbf{and} total-on \((\text{set } xs)\) \( R \) \textbf{and} \( \land x. \ x \in \text{set } xs \implies (x, x) \in R \)
\textbf{Shows} sorted-wrt \( R \) \((\text{quicksort} \ R \ \text{xs})\)
\begin{proof}
\end{proof}

\textbf{Corollary sorted-wrt-quicksort'\hspace{0.5cm}}
\textbf{Assumes} linorder-on \( A \) \( R \) \textbf{and} \( \text{set } \text{xs} \subseteq A \)
\textbf{Shows} sorted-wrt \( R \) \((\text{quicksort} \ R \ \text{xs})\)
\begin{proof}
\end{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.

\textbf{Fun quicksort' :: ('a × 'a) set ⇒ 'a list ⇒ 'a list × nat where}
\textbf{quicksort' - [] = ([], 0)}
\textbf{| quicksort' R (x # xs) = (}
\textbf{let} \( (ls', rs') = \text{partition} \ ((\lambda y. (y,x) \in R)) \ \text{xs}; \)
\textbf{\quad (ls', n1) = quicksort' R \ \text{ls};} \\
\textbf{\quad (rs', n2) = quicksort' R \ \text{rs}}
\textbf{\quad in} \\
\textbf{\quad (ls' @ [x] @ rs', length \ \text{xs} + n1 + n2))}
\end{proof}

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

\textbf{Fun qs-cost :: ('a × 'a) set ⇒ 'a list ⇒ nat where}
\textbf{qs-cost - [] = 0}
\textbf{| qs-cost R (x # xs) =}
\textbf{\quad length \ \text{xs} + qs-cost R (filter (\lambda y. (y,x)\in R) \ \text{xs}) + qs-cost R (filter (\lambda y. (y,x)\notin R) \ \text{xs})}
\end{proof}

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

\textbf{Lemma} \textbf{fst-quicksort' [simp]:} \textbf{fst \ (quicksort' \ R \ \text{xs}) = quicksort \ R \ \text{xs}}
\begin{proof}
\end{proof}

\textbf{Lemma} \textbf{snd-quicksort' [simp]:} \textbf{snd \ (quicksort' \ R \ \text{xs}) = qs-cost \ R \ \text{xs}}
\begin{proof}
\end{proof}

\section{2.2 Analysis\hspace{0.5cm}}

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):
\textbf{theorem} \textit{qs-cost-average-conv-rqs-cost}: \\
\textbf{assumes} finite \textit{A} and linorder-on \textit{B R} and \textit{A ⊆ B} \\
\textbf{shows} \quad \text{map-pmf} (\textit{qs-cost R} \text{ (pmf-of-set (permutations-of-set \textit{A}))}) = \textit{rqs-cost (card A)} \\
\textit{⟨proof⟩} \\
We therefore have the same expectation as well. (Note that we showed \textit{rqs-cost-exp n} = 2 * real (n + 1) * harm n - 4 * real n and \textit{rqs-cost-exp} \sim[sequentially] (λx. 2 * real x * ln (real x)) before. \\
\textbf{corollary} \textit{expectation-qs-cost}: \\
\textbf{assumes} finite \textit{A} and linorder-on \textit{B R} and \textit{A ⊆ B} \\
\textbf{defines} \textit{random-list} ≡ pmf-of-set (permutations-of-set \textit{A}) \\
\textbf{shows} \quad \text{measure-pmf.expectation (map-pmf (qs-cost R) \textit{random-list}) real} = \textit{rqs-cost-exp (card A)} \\
\textit{⟨proof⟩} \\
\textbf{end} \\
\textbf{References} \\