Cost Analysis of QuickSort

Manuel Eberl

April 20, 2020

Abstract

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

Contents

1 Randomised QuickSort ........................................ 2
  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

2 Average case analysis of deterministic QuickSort .... 8
  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 ⇒ '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: \(n \geq \text{length } xs\) ⇒ delete-index n xs = xs
⟨proof⟩

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

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

lemma nth-delete-index:
  \(i < \text{length } xs\) ⇒ \(n < \text{length } xs\) ⇒
  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 `trans R and total-on (set xs) R and ∀x∈set xs, (x,x) ∈ R`
assumes `(ys, n) ∈ set-pmf (rquicksort R xs)`
shows `sorted-wrt R ys ∧ mset ys = 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 :: nat ⇒ nat pmf where**

\[
\begin{align*}
\text{rqs-cost } 0 & = \text{return-pmf } 0 \\
\text{rqs-cost } (\text{Suc } n) & = \\
& \text{do } \{ i ← \text{pmf-of-set } \{\ldots n\}; a ← \text{rqs-cost } i; b ← \text{rqs-cost } (n − i); \text{return-pmf } (n + a + b) \}
\end{align*}
\]

**lemma finite-set-pmf-rqs-cost [intro!]:** finite `(set-pmf (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 ⊆ A and distinct xs`
shows `map-pmf snd (rquicksort R xs) = rqs-cost (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 :: nat ⇒ real where**

\[
\begin{align*}
\text{rqs-cost-exp } 0 & = 0 \\
\text{rqs-cost-exp } (\text{Suc } n) & = \text{real } n + (\sum_{i≤n} \text{rqs-cost-exp } i + \text{rqs-cost-exp } (n − i)) / \text{real } (\text{Suc } n)
\end{align*}
\]

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

\textbf{theorem} \textit{expectation-rqs-cost}: \textit{measure-pmf} \textit{expectation} \textit{rqs-cost} \textit{n} \textit{real} = \textit{rqs-cost-exp} \textit{n} \langle \text{proof} \rangle

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

\textbf{lemma} \textit{rqs-cost-exp-Suc}:
\textit{rqs-cost-exp} \textit{(Suc n)} = \textit{real n} + 2 / \textit{real} \textit{(Suc n)} * (\sum i \leq n. \textit{rqs-cost-exp} \textit{i}) \langle \text{proof} \rangle

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:

\textbf{theorem} \textit{rqs-cost-exp-eq} [\textit{code}]: \textit{rqs-cost-exp} \textit{n} = 2 * \textit{real} \textit{(n + 1)} * \textit{harm} \textit{n} - 4 * \textit{real n} \langle \text{proof} \rangle

\textbf{lemma} \textit{asymp-equiv-harm} [\textit{asymp-equiv-intros}]: \textit{harm} \sim \textit{at-top} (\lambda n. \textit{ln} \textit{(real n)}) \langle \text{proof} \rangle

\textbf{corollary} \textit{rqs-cost-exp-asymp-equiv} \textit{rqs-cost-exp} \sim \textit{at-top} (\lambda n. 2 * \textit{n} * \textit{ln n}) \langle \text{proof} \rangle

\textbf{lemma} \textit{harm-mono}: \textit{m} \leq \textit{n} \implies \textit{harm} \textit{m} \leq \textit{(harm} \textit{n} :: \textit{real}) \langle \text{proof} \rangle

\textbf{lemma} \textit{harm-Suc-0} [\textit{simp}]: \textit{harm} \textit{(Suc 0)} = 1 \langle \text{proof} \rangle

\textbf{lemma} \textit{harm-ge-1}: \textit{n} > 0 \implies \textit{harm} \textit{n} \geq (\textit{1}::\textit{real}) \langle \text{proof} \rangle

\textbf{lemma} \textit{mono-rqs-cost-exp}: \textit{mono} \textit{rqs-cost-exp} \langle \text{proof} \rangle

\textbf{lemma} \textit{rqs-cost-exp-leI}: \textit{m} \leq \textit{n} \implies \textit{rqs-cost-exp} \textit{m} \leq \textit{rqs-cost-exp} \textit{n} \langle \text{proof} \rangle

1.6 Version for lists with repeated elements

\textbf{definition} \textit{threeway-partition} \textbf{where}
\textit{threeway-partition} \textit{x R xs} =
(filter (\lambda y. (y,x) \in R \land (x,y) \notin R) \textit{xs},
filter (\lambda y. (x,y) \in R \land (y,x) \in R) \textit{xs},
filter (\lambda y. (x,y) \in R \land (y,x) \notin R) \textit{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.

```haskell
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 lesss :: ('a × 'a) set ⇒ 'a ⇒ '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 where
  greaters 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)
⟨proof⟩ lemma length-lesss-le [intro]: length (lesss R x xs) ≤ length xs
⟨proof⟩ lemma length-lesss-less [intro]:
  assumes x ∈ set xs
  shows length (lesss R x xs) < length xs
⟨proof⟩ 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)
⟨proof⟩ lemma length-greaters-le [intro]: length (greaters R x xs) ≤ length xs
⟨proof⟩ lemma length-greaters-less [intro]:
  assumes x ∈ set xs
  shows length (greaters R x xs) < length xs
```

6
The following function counts the comparisons made by the modified randomised Quicksort.

```ml
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)))
  )
```

(termination)

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)
  }

(terminates)

lemma finite-set-pmf-rqs'_cost [simp, intro]:
  finite (set-pmf (rqs'_cost R xs))
(termination)
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.

2 Average case analysis 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.

We can easily show that this QuickSort is correct:

We can easily show that set-quickSort is correct:

We can easily show that mset-quickSort is correct:
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
⟨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):
\begin{proof}
We therefore have the same expectation as well. (Note that we showed \( \text{rqs-cost-exp} n = 2 \times \text{real} (n + 1) \times \text{harm} n - 4 \times \text{real} n \) and \( \text{rqs-cost-exp} \sim_{\text{sequentially}} (\lambda x. 2 \times \text{real} x \times \ln (\text{real} x)) \) before.
\end{proof}

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

\end{document}

References

