Cost Analysis of QuickSort

Manuel Eberl

February 23, 2021

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


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
deleter-index [] = []
deleter-index 0 (x # xs) = xs
deleter-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 ≥ 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 rquicksort-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
- `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))**

(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
- `rqs-cost-exp 0 = 0`
- `rqs-cost-exp (Suc n) = real n + (∑ i≤n. rqs-cost-exp i + rqs-cost-exp (n − i)) / real (Suc n)`

**Lemmas rqs-cost-exp-0 = rqs-cost-exp.simps(1)**
**Lemmas rqs-cost-exp-Suc simp del = rqs-cost-exp.simps(2)**
**Lemma rqs-cost-exp-Suc-0 simp: rqs-cost-exp (Suc 0) = 0** (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.expectation} (\textit{rqs-cost} \textit{n}) real = \textit{rqs-cost-exp} \textit{n} \\
\langle 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'}:
\[ \textit{rqs-cost-exp} (\text{Suc} \textit{n}) = \text{real} \textit{n} + 2 / \text{real} (\text{Suc} \textit{n}) \ast (\sum_{i \leq n} \textit{rqs-cost-exp} \textit{i}) \]
\langle 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} [code]: \textit{rqs-cost-exp} \textit{n} = 2 \ast \text{real} (\textit{n} + 1) \ast \text{harm} \textit{n} - 4 \ast \text{real} \textit{n} \\
\langle proof\rangle

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

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

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

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

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

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

1.6 Version for lists with repeated elements

\textbf{definition} \textit{threeway-partition} \textit{where} 

\textit{threeway-partition} \textit{x} \textit{R} \textit{xs} = 
\begin{align*}
& (\text{filter} (\lambda y. (y,x) \in R \land (x,y) \notin R) \textit{xs}), \\
& (\text{filter} (\lambda y. (x,y) \in R \land (y,x) \in R) \textit{xs}), \\
& (\text{filter} (\lambda y. (x,y) \in R \land (y,x) \notin R) \textit{xs})
\end{align*}
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**

```
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 ⇒ 'a list` **where**

```
lesss R x xs = filter (λy. (y, x) ∈ R ∧ (x, y) /∈ R) xs
```

**qualified definition** `greaters` :: `(a × a) set ⇒ 'a ⇒ 'a list ⇒ 'a list` **where**

```
greaters R x xs = filter (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`

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

\[
\text{function } \text{rqs}'\text{-cost} \, :: \, ('a \times 'a \text{ set}) \Rightarrow 'a \text{ list} \Rightarrow \text{nat pmf} \text{ where}
\]
\[
\text{rqs}'\text{-cost} \, R \, xs =
\] (if \(xs = []\) then
    return-pmf 0
  else
    do
      \(i \leftarrow \text{pmf-of-set} \{..<\text{length} \, xs\}\); let \(x = xs ! i\);
      map-pmf (\(\lambda (n1,n2). \text{length} \, xs - 1 + n1 + n2\))
          (pair-pmf (\(\text{rqs}'\text{-cost} \, R \, (\text{lesss} \, R \, x \, xs)\))
                     (\(\text{rqs}'\text{-cost} \, R \, (\text{greaters} \, R \, x \, xs)\)))
    \)
  \)
\] (proof)

\textbf{termination} (proof)

\textbf{declare} \(\text{rqs}'\text{-cost} \, \text{simps} \, [\text{simp del}]\)

\textbf{lemma} \(\text{rqs}'\text{-cost-nonempty}:\)
\[
\text{xs} \neq [] \implies \text{rqs}'\text{-cost} \, R \, xs =
\] do
  \(i \leftarrow \text{pmf-of-set} \{..<\text{length} \, xs\}\);
  let \(x = xs ! i\);
  \(n1 \leftarrow \text{rqs}'\text{-cost} \, R \, (\text{lesss} \, R \, x \, xs)\);
  \(n2 \leftarrow \text{rqs}'\text{-cost} \, R \, (\text{greaters} \, R \, x \, xs)\);
  return-pmf (\(\text{length} \, xs - 1 + n1 + n2\))
\)
(proof)

\textbf{lemma} \(\text{finite-set-pmf-rqs}'\text{-cost} \, [\text{simp, intro}]:\)
\[
\text{finite} \, (\text{set-pmf} \, (\text{rqs}'\text{-cost} \, R \, xs))
\]
(proof)

\textbf{lemma} \(\text{expectation-pair-pmf-fst} \, [\text{simp}]:\)
\[
\text{fixes} \, f \, :: \, 'a \Rightarrow 'b::\{\text{banach, second-countable-topology}\}
\text{shows} \, \text{measure-pmf} \cdot \text{expectation} \, (\text{pair-pmf} \, p \, q) \, (\lambda x. f \, (\text{fst} \, x)) = \text{measure-pmf} \cdot \text{expectation} \, p \, f
\]
(proof)

\textbf{lemma} \(\text{expectation-pair-pmf-snd} \, [\text{simp}]:\)
\[
\text{fixes} \, f \, :: \, 'a \Rightarrow 'b::\{\text{banach, second-countable-topology}\}
\text{shows} \, \text{measure-pmf} \cdot \text{expectation} \, (\text{pair-pmf} \, p \, q) \, (\lambda x. f \, (\text{snd} \, x)) = \text{measure-pmf} \cdot \text{expectation} \, q \, f
\]
(proof) \textbf{lemma} \(\text{length-lesss-le-sorted}:\)
\[
\text{assumes} \, \text{sorted-wrt} \, R \, x \, i < \text{length} \, xs
\text{shows} \, \text{length} \, (\text{lesss} \, R \, (xs ! i) \, xs) \leq i
\]
lemma length-greaters-le-sorted:
assumes sorted-wrt R xs i < length xs
shows length (greaters R (xs ! i) xs) ≤ length xs − i − 1

lemma length-lesss-le':
assumes i < length xs linorder-on A R set xs ⊆ A
shows length (lesss R (insort-wrt R xs ! i) xs) ≤ i

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

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)

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 \( \forall x. x \in \text{set } xs \implies (x, x) \in R \)
  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 (λy. (y, x) \in 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) \in R) xs) + qs-cost R (filter (λy. (y, x) \not\in 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 ⊆ B
shows map-pmf (qs-cost R) (pmf-of-set (permutations-of-set A)) = rqs-cost (card A)
⟨proof⟩

We therefore have the same expectation as well. (Note that we showed rqs-cost-exp n = 2 * real (n + 1) * harm n – 4 * real n and rqs-cost-exp ∼[sequentially] (λx. 2 * real x * ln (real x)) before.

corollary expectation-qs-cost:
assumes finite A and linorder-on B R and A ⊆ B
defines random-list ≡ pmf-of-set (permutations-of-set A)
shows measure-pmf.expectation (map-pmf (qs-cost R) random-list) real = rqs-cost-exp (card A)
⟨proof⟩

end

References