Quick-Sort: A Pet Peeve

Arthur Nunes-Harwitt
Rochester Institute of Technology
Rochester, New York
anh@cs.rit.edu

Matt Gambogi
Rochester Institute of Technology
Rochester, New York
matt.gambogi@csh.rit.edu

Travis Whitaker
Alphasheets, Inc.
San Francisco, California
travis@programmable.computer

ABSTRACT
The traditional functional formulation of quick-sort is simple and elegant. But is it fast? Through a dialog, we observe that this traditional formulation does not retain certain crucial properties of the imperative version. We include a known derivation of a higher performing functional implementation together with a graph that illustrates the differences. Our pet peeve is that the faster quick-sort is frequently left out of texts on functional programming.

CCS CONCEPTS
• Applied computing → Education; • Theory of computation → Divide and conquer;

KEYWORDS
Algorithms, Quick-Sort, Functional Programming, Performance, Education

ACM Reference Format:
https://doi.org/10.1145/3159450.3159535

1 INTRODUCTION
We sat in the classroom waiting. The professor came in and stopped suddenly. He was staring at the board with a scowl. We looked too and noticed the following code [9, 13].

quicksort :: Ord α ⇒ [α] → [α]
quicksort [] = []
quicksort (p : xs) = quicksort [y | y ∈ xs, y ≤ p] ++ [p] ++ quicksort [y | y ∈ xs, y > p]

"If I see quick-sort in Haskell one more time, I’m going to be sick!" he declared.

“What’s wrong?” we asked. “Doesn’t that implementation beautifully express the essence of the quick-sort algorithm?”

“Yes and no,” replied the professor. “It certainly gets at how the sorting happens, but it doesn’t shed light on the ‘quick’ aspect. Consider the following imperative formulation [5, 8].”

def Sort(A):
Sort3(A, 0, len(A) - 1)
def Sort3(A, p, r):
    if p < r :
        q ← Partition(A, p, r)
        Sort3(A, p, q - 1)
        Sort3(A, q + 1, r)

“Notice that partitioning happens in a single pass, that there is no work involved in combining the subsequences, and that the second recursive call is in tail position. That’s why the imperative version is fast. It’s not even clear that the Haskell formulation is faster than merge-sort.”

2 COMPARISON TO MERGE-SORT
"We’ve already written merge-sort in Haskell:"

mergesort :: Ord α ⇒ [α] → [α]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = let (half1, half2) = split xs
                 in merge (mergesort half1) (mergesort half2)

split :: [α] → ([α], [α])
split xs = splita xs [] []
splita :: [α] → [α] → [α] → ([α], [α])
splita [] e o = (e, o)
splita (x : xs) e o = splita xs o (x : e)

merge :: Ord α ⇒ [α] → [α] → [α]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys)
| x < y = x : (merge xs (y : ys))
| x > y = y : (merge (x : xs) ys)
| otherwise = x : y : (merge xs ys)

"We used GHC version 8.0.1 with packages from Stackage resolver lts-7.3. A 2-core i386 Ubuntu 16.04 VM with 4 GB of memory was used as the underlying machine. The criterion benchmarking..."
suite was used to capture all result data. We found that the quicksort implementation is about 1.45 times faster than the merge-sort implementation." (In the comparison, we call the original version of quick-sort quicksort1. See Figure 1.)

“OK,” replied the professor. “But it’s not that much faster. Quicksort should be faster than that.”

3 IMPROVING PARTITION

“Improving the partition phase by reducing the number of passes is not hard. We’ve written the following one-pass function.”

\[\text{partition} :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \{\alpha\} \rightarrow \{\alpha\} \times \{\alpha\}\]

\[
\text{partition pivot lst} =
\]

\[
\begin{align*}
\text{let part } &\left[\right] \equiv \text{leq } gt \equiv (\text{leq}, gt) \\
\text{part } (x : xs) &\equiv \text{leq } gt \mid x > \text{pivot} = \text{part } xs \text{ leq } (x : gt) \\
\text{part } (x : xs) &\equiv \text{leq } gt \equiv \text{part } xs \times (x : \text{leq }) \equiv (\text{leq}, gt) \\
\text{in part } &\left[\right] [\]
\end{align*}
\]

“We make use of an accumulation parameter [6] that implicitly appends. Here is the equation for the invariant:

\[\text{qsAccum } \ell \ a = (\text{quicksort } \ell) \cdot \cdot \cdot a\]

“The base case is then simple calculation [3].”

\[\text{qsAccum } \left[\right] \ a = (\text{quicksort } [] ) \cdot \cdot \cdot a = a\]

“The recursive case is not too bad either.”

\[
\begin{align*}
\text{qsAccum } (p : x s) \ a &\equiv (\text{quicksort } (p : x s)) \cdot \cdot \cdot a \equiv (\text{quicksort } \text{leq }) \cdot \cdot \cdot (p + ((\text{quicksort } \text{gt }) \cdot \cdot \cdot a)) \equiv (\text{quicksort } \text{leq }) \cdot \cdot \cdot (p : (\text{qsAccum } \text{gt } a)) \equiv \text{qsAccum } \text{leq } (p : (\text{qsAccum } \text{gt } a))
\end{align*}
\]

where \((\text{leq}, gt) \equiv \text{partition } p x s\)

“Putting it all together, we get the following alternative functional quick-sort formulation [2, 4, 11, 12],”

\[\text{quicksort3} :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \{\alpha\} \rightarrow \{\alpha\}\]

\[
\begin{align*}
\text{quicksort3 } &\ell = \\
\text{let } &\text{qsAccum } [] \ a = a \\
\text{qsAccum } &\ell (p : x s) a = \\
\text{let } (\text{leq}, gt) &\equiv \text{partition } p x s \\
\text{in } &\text{qsAccum } \text{leq } (p : (\text{qsAccum } \text{gt } a)) \\
\text{in } &\text{qsAccum } \ell []
\end{align*}
\]

Further, this version does, in fact, perform significantly better. It is about 1.64 times faster than the previous version, it is about 2.75 times faster than the original version, and it is about 4.01 times faster than merge-sort.” (See Figure 1.)

5 CONCLUSION

“I like it,” commented the professor.

“It’s still purely functional, but the final version does partitioning in a single pass, the work for combining the subsequences involves only a small constant, and one of the recursive calls is in tail position. Further, the graph clearly shows how much faster it is than merge-sort.

“Now that I see the derivation, it is obvious. But I had assumed that it wasn’t possible since I’ve seen so many texts [1, 6, 7, 9, 10, 13] that have only the initial formulation of quick-sort. Perhaps the authors of those texts consider it a small leap, but why not publicize the good news?! I hope that in the future authors will include both formulations.”

6 ACKNOWLEDGEMENTS

We would like to thank Tim Fossum and Jim Heliotis for carefully reading previous drafts of this paper.

REFERENCES

Figure 1: Sorting performance graph