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:
Arthur Nunes-Harwitt, Matt Gambogi, and Travis Whitaker. 2018. Quick-Sort: A Pet Peeve. In *Proceedings of SIGCSE '18: The 49th ACM Technical Symposium on Computer Science Education, Baltimore , MD, USA, February 21–24, 2018 (SIGCSE '18),* 3 pages.
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[1]. 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

[1] Of course, tail-recursion does not necessarily imply a speed improvement; it only entails a space improvement.

SIGCSE '18, February 21–24, 2018, Baltimore , MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159535

suite was used to capture all result data. We found that the quick-sort 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. Quick-sort 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."

```
partition :: Ord α ⇒ α → [α] → ([α], [α])
partition pivot lst =
  let part []       leq gt               = (leq, gt)
      part (x : xs) leq gt | x > pivot = part xs leq (x : gt)
      part (x : xs) leq gt               = part xs (x : leq) gt
  in part lst [] []
```

"Using it instead of the list comprehensions yields a performance increase. The new version is about 1.67 times faster than the original version, and it is about 2.44 times faster than merge-sort." (We call this version of quick-sort *quicksort2*. See Figure 1.)

"Fine," commented the professor. "But it will be impossible for you to eliminate all the work involved when combining the results."

4 IMPROVING COMBINING

"We can completely eliminate the linear-time list append operations (++)," we replied.

"The very nature of functional programming requires that you assemble the results returned. If you are not using append, then surely you are using some other similar operation."

"It's true that we cannot eliminate all work. We use a single constant-time cons (:) operation."

"Is that all?" The professor was surprised. "That's a big improvement. How can you get away with only that?"

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

$$qsAccum\ \ell\ a = (quicksort\ \ell) +\!\!+ a$$

"The base case is then simple calculation [3]."

$$qsAccum\ []\ a = (quicksort\ []) +\!\!+ a = a$$

"The recursive case is not too bad either."

$$\begin{aligned}
&qsAccum\ (p : xs)\ a && = \\
&(quicksort\ (p : xs)) +\!\!+ a && = \\
&(quicksort\ leq) +\!\!+ ([p] +\!\!+ ((quicksort\ gt) +\!\!+ a)) && = \\
&(quicksort\ leq) +\!\!+ (p : (qsAccum\ gt\ a)) && = \\
&qsAccum\ leq\ (p : (qsAccum\ gt\ a))
\end{aligned}$$

where $(leq, gt) = partition\ p\ xs$

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

```
quicksort3 :: Ord α ⇒ [α] → [α]
quicksort3 lst =
  let qsAccum []       a = a
      qsAccum (p : xs) a =
        let (leq, gt) = partition p xs
        in qsAccum leq (p : (qsAccum gt a))
  in qsAccum lst []
```

"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

[1] Umut A. Acar and Guy E. Blelloch. 2017. *Algorithm Design: Parallel and Sequential.* http://www.parallel-algorithms-book.com/.

[2] Richard Bird. 1998. *Introduction to Functional Programming using Haskell.* Prentice Hall.

[3] Rod M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM (JACM)* 24, 1 (1977), 44–67.

[4] William Clocksin and Christopher S. Mellish. 1987. *Programming in PROLOG.* Springer.

[5] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* MIT press.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press.

[7] Brian Harvey. 1997. *Computer Science Logo Style: Beyond Programming.* Vol. 3. MIT press.

[8] Charles A. R. Hoare. 1962. Quicksort. *Comput. J.* 5, 1 (1962), 10–16.

[9] Miran Lipovača. 2011. *Learn You a Haskell for Great Good! A Beginner's Guide.* no starch press.

[10] V.S. Manis and J.J. Little. 1995. *The Schematics of Computation.* Prentice Hall. https://books.google.com/books?id=jIyIQgAACAAJ

[11] Lawrence C. Paulson. 1996. *ML for the Working Programmer.* Cambridge University Press.

[12] Fethi Rabhi and Guy Lapalme. 1999. *Algorithms: A Functional Programming Approach.* Addison-Wesley Longman Publishing Co., Inc.

[13] Simon Thompson. 2011. *Haskell: The Craft of Functional Programming.* Addison-Wesley Publishing Company.

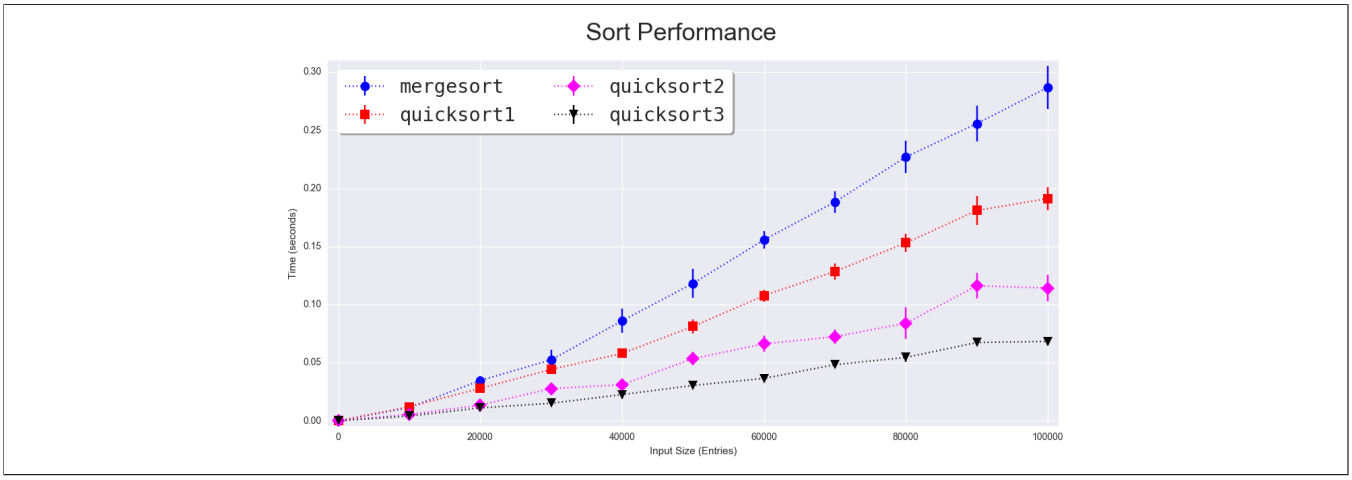

Figure 1: Sorting performance graph