Report Number: WUCS-91-20

1991-01-01

# Quicksort in Constant Space

Anne Kaldewaij and Jan Tijmen Udding

An in situ sorting algorithm, based on Quicksort, is presented with its proof of correctness. The proof illustrates a concise and elegant way to represent a sequence that has been partially sorted by Quicksort. A list representation turns out to be well suited to this purpose, and the discussion is entirely in terms of lists.

# Quicksort in Constant Space

Anne Kaldewaij
Jan Tijmen Udding

WUCS-91-20

January 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

# Quicksort in Constant Space

*Anne Kaldewaij*[*]
*Jan Tijmen Udding*
Department of Computer Science
Washington University
Campus Box 1045
St. Louis, MO 63130

February 2, 1991

**Abstract**

An *in situ* sorting algorithm, based on Quicksort, is presented with its proof of correctness. The proof illustrates a concise and elegant way to represent a sequence that has been partially sorted by Quicksort. A list representation turns out to be well suited to this purpose, and the discussion is entirely in terms of lists.

## Introduction

An efficient algorithm for sorting a sequence of integers is Quicksort [4]. It has an expected running time of $O(N \log N)$ where $N$ is the length of the sequence, although its worst case performance is quadratic. The standard Quicksort algorithm takes $O(\log N)$ space, e.g. stack space for recursion.

In [2], Durian argues that Quicksort's space requirements can be reduced to a constant at the expense of a slight (constant factor) slowdown. No attempt is made, however, to provide a proof of correctness. In [3], Gries, not convinced by an earlier attempt of Wegner [5], formally presents and proves Durian's algorithm correct. The current paper is a result of the still apparent gap between conveying the idea of the algorithm and its formal correctness proof. Moreover, as opposed to the previous attempts, we allow the sequence to be sorted to contain duplicates. We hope that our presentation of the algorithm does justice to the ingenuity of the space reduction.

[*]On leave of absence from Eindhoven University of Technology

The key issue is to capture the properties of a partially sorted sequence in a concise and tangible way. One recursively defined property turns out to capture the essence of a partially sorted sequence.

In the next section we introduce the necessary operators on lists. Subsequently, we specify procedures to swap two subsequences of a list and to partition a list. Both are used by the standard Quicksort algorithm, so we assume these to be primitive. Then we present the algorithm and its correctness proof. We conclude with a few observations.

## Lists

We present the algorithm in terms of lists of integers. For the sake of completeness we introduce some standard notations for finite lists, cf. [1]. Roman letters at the beginning of the alphabet denote elements of a list, whereas letters at the end of the alphabet denote lists. The empty list is denoted by $[\,]$, and the list consisting of $a$ by $[a]$. The catenation of two lists is denoted by juxtaposition. The list consisting of the first $i$ elements of $x$ is denoted by $x{\uparrow}i$ and the list that is the result of dropping the first $i$ elements of $x$ is denoted by $x{\downarrow}i$. We number the positions in a list from the left starting at 0, and the number of elements in list $x$ is $|x|$. The subsequence of $x$ starting at position $p$ and ending in position $q$ is denoted by $x[p..q]$. We denote the element of $x$ at position $i$ by $x.i$ and we say that $x$ is ascending, denoted by $asc.x$, if $(\forall i : 0 < i < |x| : x.(i{-}1) \leq x.i)$.

The rank of the first element of a nonempty list $x$, denoted by $rf.x$, is the number of elements in $x$ that are less than the first element of $x$. Thus, $rf.x$ is the leftmost position that the first value of $x$ takes when $x$ is sorted. The bag of values in $x$ is denoted by $[\![x]\!]$. Finally, we extend the ordering on integers to lists by saying that $x \leq y$ $(x < y)$ if all values in $x$ are at most (less than) all values in $y$. We abbreviate expressions like $x \leq y \wedge y \leq z$ to $x \leq y \leq z$. Notice that we may conclude from this that $x \leq z$ only if $y \neq [\,]$.

## Two Primitive Procedures

We use two procedures of which we give only the specifications. The first one swaps, and possibly reorders, two subsequences of a list. Since it is used in our version of Quicksort only if at least one of the two lists has length 1, it can be implemented in constant time by the usual single swap of two elements. The other one is a partitioning and can be implemented in exactly the same way as in standard Quicksort. The same pivot strategies as in standard Quicksort can be applied.

The formal specification of *swap* is that for any lists $y$, $s$, $t$, and $z$, there are lists $s'$ and $t'$ such that

$$\begin{array}{l} \{x = y\ s\ t\ z\} \\ \quad swap(x, |y|, |s|, |t|) \\ \{x = y\ t'\ s'\ z \ \wedge\ [\![s']\!] = [\![s]\!] \ \wedge\ [\![t']\!] = [\![t]\!]\}. \end{array}$$

The other procedure reshuffles the prefix of a given list and returns a position in that list, through *output* parameter $l$. To the left of $l$ everything is smaller than and to the right everything is at least the value at $l$. Formally, for any lists $u$ and $t$ there exist lists $q$ and $r$, and integer $c$ such that

$$\begin{array}{l} \{x = u\ t \ \wedge\ |u| \geq 1\} \\ \quad part(x, |u|, l) \\ \{x = q\,[c]\ r\ t \ \wedge\ |q| = l \ \wedge\ [\![q\,[c]\ r]\!] = [\![u]\!] \ \wedge\ q < [c] \leq r\} \end{array}$$

## Quicksort

We are given a sequence $h$ of integers that has to be sorted. During execution of Quicksort a prefix of the sequence has already been sorted while the remaining sequence is partitioned into a collection of segments. The partitioning into segments is ascending, i.e. all elements of each segment are at most all elements of the segments to its right. A segment itself need not yet be ascending. In each step in our implementation of Quicksort the leftmost segment is either partitioned further or added to the already sorted part of the sequence. Somehow, Quicksort has to keep track of these segments, which usually results in a nonconstant space requirement. The beauty of the algorithm discussed here is the way in which it keeps track of those segments, using the sequence itself.

The trick is to let each segment be headed by its largest element. As a result, the rank of the first element in the remaining sequence to be sorted is the length $-1$ of the first of the segments into which the sequence is partitioned, since this partitioning is ascending. To capture this property we introduce predicate *cs*, which holds for a list exactly when it is a catenation of such segments. It is formally defined as follows.

(i) $cs.[\,]$

(ii) For nonempty list $x$
$$cs.x \ \equiv\ x{\uparrow}(rf.x{+}1) \leq x{\downarrow}(rf.x{+}1) \ \wedge\ cs.(x{\downarrow}(rf.x{+}1))$$

The following property shows when the catenation of a list and a *cs*-list is a *cs*-list again. It is an immediate consequence of the definition above.

**Property 0** For lists $x$ and $y$ we have

$$x\uparrow 1 > x\downarrow 1 \ \wedge\ x \le y \ \wedge\ cs.y \quad \Rightarrow \quad cs.(x\ y)$$

□

The invariant of our program is similar to the one for Quicksort, but in addition we maintain the $cs$-property for the remainder of the list to be sorted. That is, the invariant becomes

$$P: \quad [\![h]\!] = [\![p\ x]\!] \ \wedge\ asc.p \ \wedge\ p \le x \ \wedge\ cs.x$$

This is established by $p, x := [\,], h$, provided that $cs.h$ holds. Using Property 0 and the fact that $cs.[\,]$ holds, one way to establish this is to make sure that $h$ is headed by its largest element. We can achieve this by prefixing the original sequence with a value that is larger than any other value in the sequence. From now on we assume $cs.h$.

It is clear that we are done when $x = [\,]$, thus as guard of the repetition we choose $x \ne [\,]$. In order to determine how progress can be made in the body of the repetition we abbreviate $rf.x$ to $k$ and we derive

$$\begin{array}{ll}
 & P \ \wedge\ x \ne [\,] \\
= & \quad \{ \text{ definition of } P \ \} \\
 & [\![h]\!] = [\![p\ x]\!] \ \wedge\ asc.p \ \wedge\ p \le x \ \wedge\ cs.x \ \wedge\ x \ne [\,] \\
\Rightarrow & \quad \{ \text{ definition of } cs, \text{ (ii) } \} \\
 & [\![h]\!] = [\![p\ x]\!] \wedge asc.p \wedge p \le x\uparrow(k{+}1) \le x\downarrow(k{+}1) \wedge cs.(x\downarrow(k{+}1)) \wedge x \ne [\,] \\
\Rightarrow & \quad \{ \text{ calculus and definitions of } k \text{ and rank } \} \\
 & [\![h]\!] = [\![p\ x]\!] \ \wedge\ asc.p \ \wedge\ p \le x\uparrow(k{+}1)\downarrow 1 < x\uparrow 1 \le x\downarrow(k{+}1) \ \wedge \\
 & p \le x\uparrow 1 \ \wedge\ cs.(x\downarrow(k{+}1))
\end{array}$$

Notice that $p \le x\uparrow 1$ follows from $p \le x\uparrow(k{+}1)\downarrow 1 < x\uparrow 1$ only if $k > 0$. ¿From the above derivation it is clear that if $P \ \wedge\ x \ne [\,]$ then there are lists $s$ and $t$, and integer $a$ such that

$$\begin{array}{l}
[\![h]\!] = [\![p\ x]\!] \ \wedge\ asc.p \ \wedge\ x = [a]\ s\ t \ \wedge \\
|\,s\,| = k \ \wedge\ p \le s < [a] \le t \ \wedge\ p \le [a] \ \wedge\ cs.t
\end{array} \qquad (0)$$

From (0) it is clear that in case $k = 0$ the invariant is maintained and progress is made by the statement $p, x := p\,(x\uparrow 1), x\downarrow 1$.

In case $k \ne 0$ we make progress by increasing the number of segments. We partition the first segment into two. First of all, we pledge that we shall make use of only *swap* and *part* on $x$. This means that $[\![x]\!]$ does not change and the only concern for maintaining the invariant is $cs.x$.

The set of elements that we are going to partition is $[\![s]\!]$ of (0). First we move this set to the front by swapping it with $[a]$. ¿From the specification of *swap* we conclude that after having performed $swap(x,0,1,k)$ there exists a list $u$ such that

$$x = u\,[a]\;t \;\wedge\; |u| = k \;\wedge\; u < [a] \leq t \;\wedge\; cs.t$$

Now $u$ is the segment that will be partitioned using *part*. By the specification of *part*, using $k \geq 1$, we conclude that $part(x,k,l)$ establishes

$$\begin{aligned}&x = q\,[c]\;r\,[a]\;t \;\wedge\; |q| = l \;\wedge\; |r| = k-l-1 \;\wedge\\ &q < [c] \leq r < [a] \leq t \;\wedge\; [c] < [a] \;\wedge\; cs.t\end{aligned}$$

for some lists $q$ and $r$ and integer $c$. In order to make $x$ into a $cs$-list again, we have to swap $[a]$ and $r$ and also $[c]$ and $q$. ¿From the specification of *swap* it follows that $swap(x,l+1,k-l-1,1)$ together with $swap(x,0,l,1)$ establishes

$$x = [c]\;q'\,[a]\;r'\;t \;\wedge\; q' < [c] \leq r' < [a] \leq t \;\wedge\; [c] < [a] \;\wedge\; cs.t$$

for some lists $q'$ and $r'$. Applying Property 0 twice we conclude that the resulting $x$ is a $cs$-list. Notice that we have indeed increased the number of segments. Hence, the following program solves the problem.

```
{$h\uparrow 1 > h\downarrow 1$}
$p, x := [\,], h$;
**do**  $x \neq [\,] \rightarrow$
        $k := rf.x$;
        **if**  $k = 0 \rightarrow$  $p, x := p(x\uparrow 1), x\downarrow 1$
        []  $k \neq 0 \rightarrow$  $swap(x,0,1,k)$;
                         $part(x,k,l)$;
                         $swap(x,l+1,k-l-1,1)$;
                         $swap(x,0,l,1)$
        **fi**
**od**
{$inc.p \wedge [\![p]\!] = [\![h]\!]$}
```

## Computing the rank

The above program has the same running time as standard Quicksort, provided that the statement $k := rf.x$ can be computed efficiently. From (0) we know that we have as precondition

$$x = [a]\ s\ t\ \wedge\ |s| = rf.x\ \wedge\ s < [a] \leq t$$

for some lists $s$ and $t$, and integer $a$.

Due to this partitioning of $x$, we can phrase computing $rf.x$ as computing *the* index $i$ for which $x[1..i{-}1] < x{\uparrow}1 \leq x[i..|x|{-}1]$. We can find this index with a simple linear search starting at the beginning or at the end of $x$. Starting at the beginning of $x$ results in a search of $O(rf.x)$, which is the same as the subsequent partitioning performed on the first segment, which has length $rf.x + 1$. Therefore, using a linear search for the statement $k := rf.x$ does not affect the time complexity of the algorithm.

Due to this partitioning of $x$, we can also use a binary search to find this index $i$. This would take $O(\log N)$, where $N$ is the length of the original sequence, for each time that we have to compute the rank of the first element of $x$. This is performed at most $2N$ times. Hence, also in this case the total time complexity of the algorithm is not affected.

## Conclusions

We have proved a modification of Quicksort, resulting in constant space requirements, to be correct. As opposed to our earlier attempts, this proof avoids the use of indices, yielding a much cleaner and understandable proof. It was not until we had given a name to all entities to be manipulated that we were able to reason in terms of lists. The crucial step was the introduction of the concept of the rank of a value in a list. From that point onward the proof was straightforward. This also made it clear that the presence of duplicates in the sequence did not complicate matters a great deal, as opposed to our initial feeling.

The partitioning that is commonly used in Quicksort yields three rather than two subsequences. The middle one consists of all values that are equal to the chosen pivot element. in the first subsequence all elements are smaller and in the third subsequence all elements are larger than this pivot element. For clarity of exposition we did not want to take advantage, if any, of this three-way partitioning.

## Acknowledgements

We are grateful to Coen Bron, Wim H. Hesselink, and Tom Verhoeff, who, upon reading an earlier solution that did not cater for duplicates in the sequence to be sorted, suggested solutions for the sorting problem with duplicates and urged us to present a solution that would allow duplicates. Acknowledgements are due to Kenneth C. Cox and Berry Schoenmakers for their careful reading of a draft version of this paper.

## References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, 1988.

[2] B. Durian. Quicksort without a stack. In *Proceedings of the Mathematical Foundations in Computer Science*, number 233 in Lecture Notes in Computer Science, pages 283 – 289, 1986.

[3] D. Gries. Constant-space quicksort. Unpublished, March 1989.

[4] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 2nd edition, 1975.

[5] L. Wegner. A generalized, one-way, stackless quicksort. *BIT*, 27:44 – 48, 1987.