Chapter 1

Constant-space Quicksort

David Gries

This paper appeared in A. W. Roscoe "A classical mind: essays in honour of C. A. R. Hoare". Prentice Hall International (UK) 1994 ISBN:0-13-294844-3

Introduction

It is a pleasure to write a paper that deals with algorithm *Quicksort*, which was invented by Tony Hoare, and that presents a version of *Quicksort* in terms of its correctness proof, a method of presentation that owes its existence to fundamental work by Tony Hoare. This paper is a response to [1] and [2], which attempt to describe a constant-space *Quicksort*. In [1], Durian presents his algorithm in a traditional operational fashion. The inadequacy of this approach is exemplified by the fact that his algorithm *IQS* does not work correctly for arrays of size less than 10. In [2], Wegner does provide some form of proof of correctness, but, as far as I could tell, the invariants he proposed were not invariants, and I had great difficulty understanding his algorithm.

Based on what I gleaned from [1] and [2], I developed my own version of constant-space *Quicksort*. The presentation given below is in two parts. First, I present a *Quicksort* algorithm in which variables explicitly contain the bounds of segments of the array still to be sorted. Second, I give a *coordinate transformation* to eliminate these variables, thus eliminating the extra space. This separates nicely the basic understanding of *Quicksort* from the change in data structures needed to force constant space. It also allowed me to see that a binary search could be used, and on a segment of the array that is not sorted!

Let me offer a frustration of mine. I can explain the idea behind constant-space *Quicksort* on a whiteboard in less than five minutes. However, to write a correct and understandable algorithm that incorporates that idea is not so easy, as the results of two earlier papers show. Unfortunately, ideas and their implementation in algorithms are two different things.

Experiments with constant-space *Quicksort* by Brandon Dixon found it to be about 5% slower than our best *Quicksort* on arrays of 1K to 100K elements. Since the best *Quicksort* requires only $O(log.n)$ space to sort an n-element array, it remains the algorithm of choice in most instances.

In the interests of brevity and clarity, our algorithm is only for arrays with distinct elements; the extension to arrays with duplicates is left to the reader. Finally, we take extreme care with parts of the proof, giving many details, so that the reader not too well versed in presentation-by-correctness-proof can see what really is necessary in proving a program correct.

Quicksort as a preprocessor for Insertionsort

Integer array $b[0..n-1]$, where $0 \leq n$ and all elements are distinct, can be sorted by algorithm *Insertionsort* (which appears in most introductory texts) in expected and worst-case time $O(n^2)$. However, by suitably preprocessing b in expected time $O(n \cdot log.n)$, *Insertionsort* will take only linear time, so the total expected time will be $O(n \cdot log.n)$. This preprocessing is the subject of this paper.

An integer i is called a *pivot* of b if everything to the left of $b.i$ is less than $b.i$ and everything to the right is greater:

$$pivot.i = b[..i-1] < b.i < b[i+1..] \quad . \tag{1.1}$$

We assume virtual elements $b.(-1) = -\infty$ and $b.n = \infty$, so that b has at least the pivots -1 and n. Suppose i and j are pivots satisfying $i < j \leq i + H$ for some constant $H > 1$, so that

$$b[..i] < b[i+1..j-1] < b[j..] \quad .$$

Then, when sorting b, *Insertionsort* requires at most time $O(H)$ to place one element of $b[i+1..j-1]$ in its final position and time $O(H \cdot (j-i-1))$ to place them all. Also, all pivots are in their final position and require only constant-time processing. Hence, if successive pivots of b are at most H apart, *Insertionsort* requires at most time $O(H \cdot n)$ to sort b. Thus, we seek preprocessing that creates such successive pivots.

Linear-space preprocessing

We now present an algorithm for permuting $b[0..n-1]$ so that its successive pivots are at most H apart. With B denoting the initial value of b, and remembering the two virtual elements $b.(-1)$ and $b.n$, the precondition is

$$\begin{aligned} Q:\ & 0 \leq n \ \wedge\ b[-1..n] = B \ \wedge \\ & b.-1 = -\infty \ \wedge\ b.n = \infty \ \wedge \\ & (\forall i, j \mid -1 \leq i < j \leq n : b.i \neq b.j) \quad . \end{aligned}$$

In presenting the postcondition, we use the following notation and terminology on sequences. Juxtaposition denotes catenation of sequences and elements. When they are not explicitly typed, capital letters denote elements and small letters sequences. Function $\mathit{first}.x$ yields the first element of sequence x and $\mathit{last}.x$ its last, and function $\mathit{tail}.x$ is defined by $x = \mathit{first}.x\ \mathit{tail}.x$. Finally, for sequences x and y we define

$$x \ \mathbf{seg}\ y = (\exists w, z \mid : w\ x\ z = y) \quad .$$

The postcondition of the preprocessing algorithm consists of the conjuncts $R0$, $R1$, $R2$, and $R3$, given below, which use two variables u and v of type $\mathit{seq}(\mathit{int})$ to contain sequences of pivots. Two variables are used instead of one to ease the later exposition.

$R0$ states that b is a permutation of its initial value. $R1$ states that sequence $u\ v$ is in strictly ascending order ($\mathit{increasing}(u\ v)$) and defines the first element of u and last element of v as the virtual pivots of b . $R2$ indicates that all elements of $u\ v$ are pivots. Finally, $R3$ indicates that successive elements of $u\ v$ are at most H apart.

$$\begin{array}{ll} R0: & \mathit{perm}(b, B) \\ R1: & \mathit{increasing}(u\ v) \wedge \mathit{first}.u = -1 \wedge \mathit{last}.v = n \\ R2: & (\forall I \mid I \in (u\ v) : \mathit{pivot}.I) \\ R3: & (\forall I, J \mid (I\ J)\ \mathbf{seg}\ (u\ v) : J - I \leq H) \end{array}$$

Algorithm (1.2) below truthifies this postcondition.

$$\begin{array}{l} u, v := \langle -1 \rangle, \langle n \rangle; \\ \{\text{Invariant} : R0 \wedge R1 \wedge R2 \wedge P3 \text{ (see below)}\} \\ \mathbf{do}\ \mathit{first}.v - \mathit{last}.u > H \rightarrow \\ \qquad \mathbf{var}\ k : \mathit{int}; \\ \qquad \mathit{Partition}(b, \mathit{last}.u + 1, \mathit{first}.v - 1, k); \\ \qquad \{R0 \wedge R1 \wedge R2 \wedge P3 \wedge P4 \text{ (see below)}\} \\ \qquad v := k\ v \\ \ [\!]\ \mathit{first}.v - \mathit{last}.u \leq H \wedge \mathit{first}.v \neq n \rightarrow u, v := u\ \mathit{first}.v, \mathit{tail}.v \\ \mathbf{od} \end{array} \tag{1.2}$$

The invariant is similar to postcondition R , except that $R3$ has been weakened to $P3$, which says that successive pivots in u (and not $u\ v$) are at most H apart).

$$P3: \ (\forall I, J \mid (I\ J)\ \mathbf{seg}\ u : J - I \leq H) \tag{1.3}$$

$$P4: \ \mathit{last}.u < k < \mathit{first}.v \wedge b[..k-1] < b.k < b[k+1..] \tag{1.4}$$

Statement $Partition(\ldots)$ of the algorithm is the standard partition algorithm, which permutes nonempty segment $b[last.u+1..first.v-1]$ and sets k to truthify

$$\begin{aligned}&last.u < k < first.v \;\wedge \\ &b[last.u+1..k-1] < b.k < b[k+1..first.v-1] \quad .\end{aligned} \tag{1.5}$$

Partition does not falsify the invariants $R0$, $R1$, $R2$, and $P3$. Predicate $P4$ in the postcondition of *Partition* follows from (1.5) and $R2$. Thus, execution of $Partition(\ldots)$ creates a new pivot k. We shall not deal further with $Partition(\ldots)$.

We now deal with the initial truth and the invariance of $R0$, $R1$, $R2$, and $P3$. $R0$ is initially true, since initially $b = B$. The only statement that changes b is $Partition(\ldots)$, and it is guaranteed only to permute b, so $R0$ remains true.

Consider $R1$. With $u = [-1]$ and $v = [n]$, and with $n \geq 0$, $R1$ is initially true. We now show the invariance of $R1$ with regard to the statement $v := k\ v$ of the first guarded command. We have,

$$\begin{aligned}&\quad wp(\text{`}v := k\ v\text{'},\ R1)\\ =&\quad\quad \langle\text{Definition of } wp \text{ and } R1\rangle\\ &\quad increasing(u\ k\ v) \wedge first.u = -1 \wedge last.(k\ v) = n \quad .\end{aligned}$$

The first conjunct follows from $P4$ and $R1$, and the last two conjuncts follow from $R1$. Hence, $R1$ is true after execution of the first guarded command.

Consider the invariance of $R1$ over the second guarded command. We have

$$\begin{aligned}&\quad wp(\text{`}u, v := u\ first.v, tail.v\text{'},\ R1)\\ =&\quad\quad \langle\text{Definition of } wp \text{ and } R1\rangle\\ &\quad increasing(u\ first.v\ tail.v) \wedge\\ &\quad first.(u\ first.v) = -1 \wedge last.(tail.v) = n\\ =&\quad\quad \langle\text{Definition of } first \text{ and } tail\rangle\\ &\quad increasing(u\ v) \wedge first.(u\ first.v) = -1 \wedge last.(tail.v) = n\end{aligned}$$

The first two conjuncts are implied by $R1$. The last conjunct holds from $R1$ provided that $tail.v$ is not empty, which follows from the guard $first.v \neq n$ and conjunct $last.v = n$ of $R1$.

The proofs of invariance of $R2$ and $P3$ are just as simple and can be easily verified informally as well as formally. Hence, they are left to the reader.

So, $R0$, $R1$, $R2$, and $P3$ are loop invariants and are true upon termination. We now prove that upon termination $R3$ holds.

$$\begin{aligned}&\quad P3 \wedge \text{falsity of the loop guards}\\ =&\quad\quad \langle\text{Predicate calculus}\rangle\\ &\quad P3 \wedge first.v - last.u \leq H \wedge first.v = n\\ =&\quad\quad \langle R1\rangle\\ &\quad P3 \wedge first.v - last.u \leq H \wedge v = [n]\\ =&\quad\quad \langle\text{Predicate calculus; One-point rule, } (\forall x \mid x = e : P) \equiv P[x := e]\rangle\end{aligned}$$

$$
\begin{array}{ll}
 & P3 \wedge (\forall I, J \mid (I\ J)\ \mathbf{seg}\ (last.u\ v) : J - I \le H) \\
= & \quad \langle \text{Range-split; Definition of } R3 \rangle \\
 & R3
\end{array}
$$

To prove termination of the loop, consider the expression $(\#u + \#v) + \#u$. By $R1$, it is bounded above by $2 \cdot n + 3$. Its initial value is 3. Each iteration increases it by 1. Hence, there are at most $2 \cdot n$ iterations.

Algorithm (1.2) is a version of *Quicksort*, as a preprocessor for *Insertionsort*, in which the sequence of generated pivots is maintained explicitly and the segments of b delimited by those pivots are processed in left-to-right order. The analysis of its expected and worst-case execution times is the same as that of other versions of *Quicksort* and is left to the reader. Our next step is to provide a coordinate transformation that eliminates variables u and v, leaving a constant-space *Quicksort*.

A coordinate transformation

Our coordinate transformation replaces variables b, u, v by an array c and integer variables i, j, h. Initially, we assume that $b = c$, and we prove that, upon termination of the transformed algorithm, again $b = c$. Thus, the final algorithm *Quicksort*s c and not b. The relationship between b, u, v and c, i, j, h is given by three coupling invariants $C0$, $C1$, and $C2$. The first coupling invariant defines i, j and h in terms of u and v:

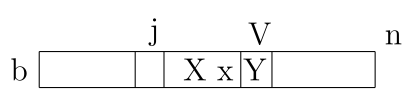
$$C0 : \ i = last.u \wedge j = first.v \wedge (h\ n)\ \mathbf{seg}\ (u\ v)$$

$C0$ allows replacement expressions $last.u$ and $first.v$ of algorithm (1.2) by i and j. Variable h is the penultimate pivot in sequence $u\ v$. It is needed because the segment $b[h+1..n-1]$ has to be handled specially because its delimiting pivot n references the virtual array element $b.n$ instead of a real array element.

The second coupling invariant indicates the segments of b and c that are equal:

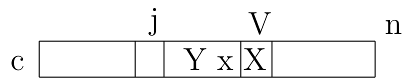
$$C1 : \ c[..j] = b[..j] \wedge c[h+1..] = b[h+1..] \quad .$$

It remains to define $c.t$ for $j < t \le h$. This definition is the key to achieving a constant-space *Quicksort*. Suppose $\#v > 2$, and introduce V and $\overline{v}$ satisfying $v = j\ V\ \overline{v}$. Note that $V \le h$. Consider the second guarded command of algorithm (1.2), which sets v to $tail.v$. In order to maintain $C0$ when $v := tail.v$ is executed, j has to be set to V, but without referring to v! We make this possible by defining $c[j+1..V]$ appropriately. Suppose $b[j+1..V]$ has the following form:



where $X < x < Y$

where x is a sequence and X and Y are elements. Then, $c[j+1..V]$ has the value shown below:



where $X < x < Y$

Thus, $c[j+1..V]$ is $b[j+1..V]$ with its first and last elements swapped. Since $b[k] \le b.V$ for $j < k \le V$, we have $c[j+1..V] \le c.(j+1)$. Since $b.V < b[V+1..]$, we have $c.(j+1) < c[V+1..]$. Hence, given j, we can determine V using a linear search (or as we shall see later, a binary search), as the single integer satisfying

$$j < V \le h \wedge c[j+1..V] \le c.(j+1) < c[V+1..] \quad .$$

The above analysis gives the main idea behind constant-space *Quicksort*, and if that is all the reader wants they may stop here.

The above analysis, with the pictures, deals poorly with the case $j+1 = V$. We now introduce notation to deal rigorously with the definition of $c[j+1..V]$. Define function $sw(b,p,q)$, for b an array (segment) and p, q integers, to be

$$sw(b,p,q) = \text{``}b \text{ with } b.p \text{ and } b.q \text{ swapped''} \quad .$$

Then define $c[j+1..V]$ to be $b[j+1..V]$ with its first and last elements swapped:

$$c[j+1..V] = sw(b[j+1..V], j+1, V)$$

Using this notation, we define all segments of c that are delimited by pivots in v, except the last:

$$C2: \ (\forall I, J \mid (I\ J) \textbf{ seg } v \wedge J \ne n : c[I+1..J] = sw(b[I+1..J], I+1, J) \quad .$$

With this definition of $c[j+1..V]$, the following theorem shows how to find the follower of j in v.

Theorem 0. Given $R1$, $R2$, $C0$, $C1$, $C2$, and $j < h$, the follower of j in v is the unique solution of $f.t$ defined by

$$f.t \equiv j < t \le h \wedge c.t \le c.(j+1) < c.(t+1) \quad . \qquad (1.6)$$

Proof. First, we prove that $f.V$ holds for the follower V of j in v. Since $j < V \le h$, by the definition of c there exists a value p, $V < p$, such that $b.p = c.(V+1)$. Using this value p, we prove that the second conjunct of $f.V$ holds:

$$\begin{array}{ll}
 & c.V \\
= & \quad \langle \text{Definition of } c \rangle \\
 & b.(j+1) \\
\le & \quad \langle V \text{ is a pivot and } j+1 \le V \rangle \\
 & b.V \\
= & \quad \langle \text{Definition of } c \rangle \\
 & c.(j+1) \qquad \text{—hence, } c.V \le c.(j+1) \\
= & \quad \langle \text{Definition of } c \rangle \\
 & b.V
\end{array}$$

$$
\begin{array}{ll}
< & \quad \langle V \text{ is a pivot of } b \text{ and } V < p \rangle \\
 & b.p \\
< & \quad \langle \text{Definition of } c \rangle \\
 & c.(V+1) \qquad \text{—hence, } c.V \le c.(j+1) < c.(V+1)
\end{array}
$$

Hence, $f.V$ is true. Now consider any value t, $j+1 \le t < V$. Since $b[j+1..V] \le b.V$, we have $c[j+1..V] \le c.(j+1)$. Hence, $c(t+1) \le c.(j+1)$, so $f.t$ is false.

Finally, consider a value t satisfying $V < t \le h$. Since V is a pivot, $b.V < b.t$. By the definition of c, $c.(j+1) < c.t$, so $f.t$ is false.

Thus, the follower of j in v is the only solution of f defined by the equation of the theorem. Q.E.D.

We now provide translations of expressions and statements of (1.2). For each expression in u, v, b, we provide an equal expression in i, j, h, c, with the coupling invariants being used to prove equality. For the initialization of u, v we provide initialization for i, j, h such that the simultaneous execution of the initializations truthifies the coupling invariants. Finally, for each other statement of (1.2) that involves u, v, b, we provide an equivalent statement that involves i, j, h, c, such that the coupling invariants are invariantly true over the simultaneous execution of the two statements.

The translation of $\mathit{last}.u$ is i (by $C0$).

The translation of $\mathit{first}.v$ is j (by $C0$).

The translation of $u, v := [-1], [n]$ is $i, j, h := -1, n, -1$. That the two together truthify $C0$ is trivial. $C1$ is truthified because initially $b = c$ is assumed. $C2$ is truthified, with the range of quantification empty.

The translation of $Partition(b,\ \mathit{last}.u+1,\ \mathit{first}.v-1,\ k)$ is $Partition(c, i+1, j-1, k)$. By $C0$ and $C1$, the two segments of b and c that are being partitioned have the same value, so that executing the two calls leave the segments still equal; further the two calls store the same value in k.

The translation of $v := k\ v$ is

$$
\begin{array}{l}
\textbf{if } i = h \rightarrow j, h := k, k \\
[]\ i < h \rightarrow \text{Swap } c.(k+1), c.j;\ j := k \\
\textbf{fi}
\end{array}
$$

We prove that this translation is correct. First, $C0 \Rightarrow i \le h$, so the statement does not abort. We handle the cases $i = h$ and $i < h$ separately. Assume $i = h$, i.e. we deal with the guarded command $j, h := k, k$. We prove that $C0$, $C1$, and $C2$ are maintained by it.

$$
\begin{array}{ll}
 & wp(\text{`}v := k\ v; j, h := k, k\text{'}, C0) \\
= & \quad \langle \text{Definition of } wp \text{ and } C0 \rangle \\
 & i = \mathit{last}.u \wedge k = \mathit{first}.(k\ v) \wedge (k\ n)\ \textbf{seg}\ (u\ k\ v)
\end{array}
$$

The first conjunct is true by $R1$; the second by the definition of *first*. Consider the third. From $i = h$, $C0$, and $R1$, we have $v = [n]$, from which the third conjunct follows.

$$\begin{array}{ll} & wp(\text{`}v := k\ v; j, h := k, k\text{'}, C1) \\ = & \quad \langle \text{Definition of } wp \text{ and } C1 \rangle \\ & c[..k] = b[..k] \wedge c[k+1..] = b[k+1..] \\ = & \quad \langle \text{Property of arrays} \rangle \\ & c = b \end{array}$$

From $C0$, $i = h$, and $R1$, we have $j = n$. Together with $C1$, this implies $b = c$. Hence, $C1$ is maintained by the first guarded command.

Similarly, $C2$ remains true —with its range of quantification empty.

Now consider the case $i < h$, i.e. consider the second guarded command.

$$\begin{array}{ll} & wp(\text{`}v := k\ v\,;\ \text{Swap } c.(k+1), c.j\,;\ j := k\text{'},\ C0) \\ = & i = last.u \wedge k = first.(k\ v) \wedge (h\ n)\ \mathbf{seg}\ (u\ k\ v) \end{array}$$

The first conjunct is in $R1$, the second is true by the definition of *first*, and the third follows from $R1$ and $i < h$.

$$\begin{array}{ll} & wp(\text{`}v := k\ v;\ \text{Swap } c.(k+1), c.j;\ j := k\text{'},\ C1) \\ = & \quad \langle \text{Definition of } wp \text{ and } C1 \rangle \\ & sw(c, k+1, j)[..k] = b[..k] \wedge sw(c, k+1, j)[h+1..] = b[h+1..] \\ = & \quad \langle \text{By } P4 \text{ and } C0,\ i < k < j \le h \rangle \\ & c[..k] = b[..k] \wedge c[h+1..] = b[h+1..] \end{array}$$

This follows from $k < j$ and $C1$.

$$\begin{array}{ll} & wp(\text{`}v := k\ v;\ \ \text{Swap } c.(k+1), c.j;\ j := k\text{'}, C2) \\ = & \quad \langle \text{Definition of } wp \text{ and } C2 \rangle \\ & (\forall I, J \mid (I\ J)\ \mathbf{seg}\ (k\ v) \wedge J \ne n : sw(c, k+1, j)[I+1..J] = \\ & \qquad\qquad\qquad\qquad\qquad\qquad sw(b[I+1..J], I+1, J)) \\ = & \quad \langle \text{Range split: One-point rule; } j = first.v \rangle \\ & sw(c, k+1, j)[k+1..j] = sw(b[k+1..j], k+1, j) \wedge C2 \\ = & \quad \langle \text{Interchange of swapping and referencing a subsegement} \rangle \\ & sw(c[k+1..j], k+1, j) = sw(b[k+1..j], k+1, j) \wedge C2 \\ = & \quad \langle \text{Property of swap} \rangle \\ & c[k+1..j] = b[k+1..j] \wedge C2 \end{array}$$

The first conjunct follows from $C1$.

The translation of $u, v := u\ first.v, tail.v$ is

$$\begin{array}{l} i := j; \\ \mathbf{if}\ i = h \rightarrow\ \ j := n \\ [\!]\ \ i < h \rightarrow\ \ j := V\text{, where } V \text{ satisfies } f.V \text{ (see (1.6))}; \\ \qquad\qquad\quad \text{Swap } c.(i+1), c.j \\ \mathbf{fi} \end{array}$$

The proof of correctness of this replacement is similar to that of the previous replacement and is left to the reader. The key, of course, is Theorem 0.

Using binary search

Even though $c[j+1..h+1]$ is not sorted, binary search can be used to find V that satisfies $f.V$ (see (1.6)), as required in the translation of $u, v := u\ \mathit{first}.v, \mathit{tail}.v$. Here, without much explanation, is a binary search algorithm due to E.W. Dijkstra, written as a procedure:

{Let initially $p = P$ and $q = Q$. Given is $c.p \le x < c.q$. Set p to truthify the following, while referencing only elements of $c[P+1..Q-1]$:
    $P \le p < Q \ \wedge\ c.p \le x < c.(p+1)$}
procedure *bsearch*(**var** b:**array** ; **var** p:*int*; q, x:*int*);
{invariant: $P \le p < q \le Q \ \wedge\ c.p \le x < c.q$}
do $p+1 \ne q \rightarrow$ **var** $e := (p+q) \div 2$;
    $\{P \le p < e < q \le Q\}$
    if $c.e \le x \rightarrow p := e$
    $[\!]$ $c.e > x \rightarrow q := e$
    fi
od

This algorithm is readily seen to satisfy its specification, though $c[P..Q]$ need not be ordered. Since the value V that satisfies (1.6) is unique, the following statement stores the desired value V in j:

$$j := i+1;\ \mathit{bsearch}(c, j, h+1, c.(i+1)) \quad . \tag{1.7}$$

The final algorithm

Making the replacements in algorithm (1.2) that are described in the previous section yields the following algorithm. It would be nice to have a program construct to perform such a coordinate transformation automatically, given the statements and expressions to be replaced and their replacements.

var $i, j, h := -1, n, -1$;
do $j - i > H \rightarrow$ **var** k;
    $\mathit{Partition}(c, i+1, j-1, k)$;
    if $i = h \rightarrow j, h := k, k$
    $[\!]$ $i < h \rightarrow$ Swap $c.(k+1), c.j$; $j := k$
    fi
$[\!]$ $j - i \le H \ \wedge\ j \ne n \rightarrow i := j$;
    if $i = h \rightarrow j := n$

$[]\ i < h \rightarrow j := i+1;\ bsearch(c, j, h+1, c.(i+1));$
$\qquad\text{Swap } c.(i+1), c.j$

fi

od

Note that, from the falsity of the guards of the loop, we have $j = n$; together with $C1$, this implies $b = c$.

This algorithm requires only constant extra space. In return, for each pivot created it requires two swaps and a binary search of $b[j..h]$, which takes at most logarithmic time. Since there are at most $n+2$ pivots, the algorithm remains $O(n \cdot log.n)$.

Acknowledgements

This work was supported by NSF under grant IRI-8804801 and by NSF and Darpa under grant ASC-8800465. Brandon Dixon spent some time analyzing constant-space *Quicksort* under the NSF program Research Experience for Undergraduates. Thanks go to Edsger W. Dijkstra for advising me to make the presentation rigorous.

References

[1] Durian, B. Quicksort without a stack. *Proc.Math.Found. Computer Science*, LNCS 233, August 1986, 283-289.

[2] Wegner, L.M. A generalized, one-way, stackless quicksort. *BIT 27* (1987), 44-48.

[3] Gries, D. *The Science of Programming.* Springer Verlag, New York, 1981.