Derivation of a Sorting Algorithm

Martin Ward
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE

February 25, 1999

1 Transformations

Theorem 1 *Proving Termination:* If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is a term giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if

$$\forall t_0.\,((\mathbf{P} \ \wedge\ \mathbf{t} \preccurlyeq t_0) \Rightarrow \mathrm{WP}(\mathbf{S}[\{\mathbf{P} \ \wedge\ \mathbf{t} \prec t_0\}/X], \mathbf{true})) \tag{1}$$

then $\mathbf{P} \Rightarrow \mathrm{WP}(\underline{\mathbf{proc}}\ X\ \equiv\ \mathbf{S}., \mathbf{true})$.

Theorem 2 If $\preccurlyeq$ is a well-founded partial order on some set Γ and $\mathbf{t}$ is a term giving values in Γ and t_0 is a variable which does not occur in $\mathbf{S}$ then if

$$\forall t_0.\,((\mathbf{P} \ \wedge\ \mathbf{t} \preccurlyeq t_0) \Rightarrow \mathbf{S}'\ \leq\ \mathbf{S}[\{\mathbf{P} \ \wedge\ \mathbf{t} \prec t_0\};\ \mathbf{S}'/X]) \tag{2}$$

then $\mathbf{P} \Rightarrow \mathbf{S}'\ \leq\ \underline{\mathbf{proc}}\ X\ \equiv\ \mathbf{S}.$

Theorem 3 *If* $\Delta \vdash \mathbf{S}_1\ \leq\ \mathbf{S}$ *and* $\Delta \vdash \mathbf{S}_2\ \leq\ \mathbf{S}$ *then* $\Delta \vdash \underline{\mathbf{join}}\ \mathbf{S}_1\ \sqcup\ \mathbf{S}_2\ \underline{\mathbf{nioj}}\ \leq\ \mathbf{S}$.

2 An Example of an Algorithm Derivation

In this section we sketch the derivation of a version of Hoare's Quicksort algorithm [3]. This illustrates the use of the **join** construct in writing concise abstract specifications which can be transformed into efficient algorithms. It also illustrates the application of the theorem on recursive implementation of statements (Theorem 2) and the refinement rules.

2.1 Notation

We use $a\,..\,b$ to represent the sequence of integers from a to b inclusive. If A is an array then $A[a\,..\,b]$ represents the sequence of array elements $A[a]$ to $A[b]$ inclusive. On the left hand side of an assignment it indicates that A takes on a new value with only those elements changed. If m is less than every element of $A[a\,..\,b]$ we write $m < A[a\,..\,b]$. Similarly if each element of $A[a'\,..\,b']$ is less than every element of $A[a\,..\,b]$ we write $A[a'\,..\,b'] < A[a\,..\,b]$. The array A is sorted if each element is less than or equal to the next ie:

$$\mathit{sorted}(A[a\,..\,b])\ =_{\mathrm{DF}}\ \forall a \leqslant i < b.\, A[i] \leqslant A[i+1]$$

An array A is a permutation of A' if they contain the same elements in a different order, ie:

$$\mathit{perm}(A[a\,..\,b], A'[a\,..\,b])\ =_{\mathrm{DF}}\ \exists \pi\colon a\,..\,b\ \rightarrow\ a\,..\,b.\, \forall a \leqslant i \leqslant b.\, A[i] = A'[\pi(i)]$$

2.2 The Specification

With this notation the specification of a sorting program is simple. We want a program which leaves array A sorted and which achieves this end by carrying out a permutation of the elements of the array. Thus the specification of a program which sorts $A[a\,..\,b]$ may be written:

$$SORT(a,b) \;=_{\text{DF}}\; \underline{\mathbf{join}}\; SORTED(a,b) \sqcup PERM(a,b)\; \underline{\mathbf{nioj}}$$

where

$$SORTED(a,b) \;=_{\text{DF}}\; \langle A[a\,..\,b]\rangle/\langle\rangle.sorted(A[a\,..\,b])$$

and

$$PERM(a,b) \;=_{\text{DF}}\; A[a\,..\,b] := A'[a\,..\,b].perm(A[a\,..\,b], A'[a\,..\,b])$$

This concisely expresses what a sorting algorithm is to achieve without being biased to any particular implementation (the direct translation: "try every possible sorted array until one is found which is a permutation of the original" cannot be regarded as an algorithm—particularly if the set of possible element values is infinite!). An important thing to note about this specification is that Theorem 3 provides a simple way to test the correctness of a possible implementation: test that it refines both components of the **join**.

So one way (not necessarily the clearest!) to write a program to sort the array $A[a\,..\,b]$ where $b \geqslant a$ is:

$$\begin{array}{lll}
SORT(a,b) \;\equiv \underline{\mathbf{join}}\; \underline{\mathbf{for}}\; i := a\; \underline{\mathbf{to}}\; b\; \underline{\mathbf{step}}\; 1\; \underline{\mathbf{do}} & & \\
\qquad\qquad\qquad \langle j\rangle/\langle\rangle.(a \leqslant j \leqslant n); & & \\
\qquad\qquad\qquad \langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle\; \underline{\mathbf{od}} & & \text{Swap } A[j] \text{ and } A[i] \\
\qquad\quad \sqcup \langle x\rangle/\langle\rangle.\mathbf{true}; & & \text{Pick any value for } x \\
\qquad\qquad A[a] := x; & & \\
\qquad\qquad \underline{\mathbf{for}}\; i := a+1\; \underline{\mathbf{to}}\; b\; \underline{\mathbf{step}}\; 1\; \underline{\mathbf{do}} & & \\
\qquad\qquad\qquad \langle x\rangle/\langle\rangle.(x > A[i-1]); & & \text{Pick a value bigger than } A[i-1] \\
\qquad\qquad\qquad A[i] := x\; \underline{\mathbf{od}}\; \underline{\mathbf{nioj}}. & &
\end{array}$$

2.3 Quicksort

The Quicksort algorithm was described by Hoare in [3]. The basic idea is to divide the array into two components which can be sorted separately resulting in a sorted array. Thus we have the following outline:

$$\begin{array}{l}
QSORT(a,b) = \underline{\mathbf{begin}}\; p : \langle A[a\,..\,b], p\rangle := \langle A'[a\,..\,b], p'\rangle.\mathbf{Q}; \\
\qquad\qquad\qquad SORT(a, p-1);\; SORT(p+1, b)\; \underline{\mathbf{end}}
\end{array}$$

We need to determine $\mathbf{Q}$: ie what to assign to A and p so that $QSORT$ is a refinement of $SORT$. Since $SORT(a,b)$ is a refinement of $PERM(a,b)$, the same must be true of $QSORT$. Thus the assignment involving $\mathbf{Q}$, followed by two refinements of $PERM$ is also a refinement of $PERM$. This means that this assignment must also be a refinement of $PERM'$ (a modification of $PERM$ which also assigns to p), so we can join $PERM'(a,b)$ to the first component of $QSORT$ thus:

$$\begin{array}{l}
QSORT(a,b) = \\
\underline{\mathbf{begin}}\; p : \underline{\mathbf{join}}\; PERM'(a,b) \\
\qquad\qquad \sqcup \langle A[a\,..\,b], p\rangle := \langle A'[a\,..\,b], p'\rangle.\mathbf{Q}\; \underline{\mathbf{nioj}}; \\
\quad SORT(a, p-1);\; SORT(p+1, b)\; \underline{\mathbf{end}}
\end{array}$$

where

$$PERM'(a,b) \;=_{\text{DF}}\; \langle A[a\,..\,b], p\rangle := \langle A'[a\,..\,b], p'\rangle.perm(A[a\,..\,b], A'[a\,..\,b])$$

Turning to the other component of $SORT$ we note that A can only finish sorted if after the **join** in $QSORT$ we have: $A[a\,..\,p-1] \leqslant A[p] \leqslant A[p+1\,..\,b]$ (this relation is invariant over the inner $SORT$s so it must hold after the **join** statement). Thus we arrive at the following definition of $QSORT$:

$QSORT(a, b) =$
$\underline{\mathbf{begin}}\ p:\underline{\mathbf{join}}\ \langle A[a \mathinner{..} b], p\rangle := \langle A'[a \mathinner{..} b], p\rangle.perm(A[a \mathinner{..} b], A'[a \mathinner{..} b])$
$\qquad\qquad \sqcup \langle A[a \mathinner{..} b], p\rangle := \langle A'[a \mathinner{..} b], p\rangle.(A'[a \mathinner{..} p'-1] \leqslant A'[p'] \leqslant A'[p'+1 \mathinner{..} b])\ \underline{\mathbf{nioj}};$
$\quad SORT(a, p-1);\ SORT(p+1, b)\ \underline{\mathbf{end}}$

To formally verify that this is a correct refinement of $SORT$:

- Prove that it refines $PERM(a, b)$. This follows because $SORT$ refines $PERM$ and a sequence of $PERM$ statements is equivalent to one $PERM$;
- Prove that it refines $SORTED(a, b)$ This follows because:
  1. The **join** establishes $A[a \mathinner{..} p-1] \leqslant A[p] \leqslant A[p+1 \mathinner{..} b]$;
  2. This relation is preserved by $SORT(a, p-1)$ and $SORT(p+1, b)$ since it is preserved by $PERM(a, p-1)$ and $PERM(p+1, b)$ (which are defined in the obvious way);
  3. $SORT(a, p-1)$ establishes $sorted(A[a \mathinner{..} p-1])$ and $SORT(p+1, b)$ establishes $sorted(A[p+1 \mathinner{..} b])$;

  these three relations together establish $sorted(A[a, b])$;

Thus $QSORT(a, b)$ refines both components of $SORT(a, b)$ and therefore by Theorem 3 it is a refinement of $SORT(a, b)$ itself.

At the moment $QSORT$ is still defined in terms of $SORT$. However, if we take out the case $a \geqslant b$ (when the array is already sorted) we can use the theorem on recursive implementation of specifications (Theorem 2) to automatically derive a recursive version:

$\underline{\mathbf{proc}}\ QSORT(a, b)\ \equiv$
$\underline{\mathbf{if}}\ b > a$
$\quad \underline{\mathbf{then}}\ \underline{\mathbf{begin}}\ p:\underline{\mathbf{join}}\ \langle A[a \mathinner{..} b], p\rangle := \langle A'[a \mathinner{..} b], p'\rangle.perm(A[a \mathinner{..} b], A'[a \mathinner{..} b])$
$\qquad\qquad\qquad \sqcup \langle A[a \mathinner{..} b], p\rangle/\langle\rangle.a \leqslant p \leqslant b \wedge (A[a \mathinner{..} p-1] \leqslant A[p] \leqslant A[p+1 \mathinner{..} b])\ \underline{\mathbf{nioj}};$
$\qquad\quad QSORT(a, p-1);\ QSORT(p+1, b)\ \underline{\mathbf{end}}\ \underline{\mathbf{fi}}.$

This program is closer to an implementation, but still contains a **join** construct to which we now turn our attention. The effect of this construct is to permute A and assign some value to p so that the relation $A[a \mathinner{..} p-1] \leqslant A[p] \leqslant A[p+1 \mathinner{..} b]$ is satisfied.

We will implement this using a loop: for the loop invariant we weaken the condition by introducing two variables i, j to replace p. (This technique of "weakening the required condition" is discussed by Gries in [2]). We arbitrarily choose the element $A[b]$ to be the element which ends up in $A[p]$. Our loop invariant is: $A[a \mathinner{..} i-1] \leqslant A[b] \leqslant A[j+1 \mathinner{..} b]$ The terminating condition will be $i > j$. On termination we can swap $A[j+1]$ and $A[b]$ and set $p := j+1$ and our required relation will be satisfied. The invariant is easily initialised with the assignments $i := a;\ j := b-1$. We have the following loop:

$\underline{\mathbf{begin}}\ i := a,\ j := b-1:$
$\quad \underline{\mathbf{while}}\ i \leqslant j\ \underline{\mathbf{do}}$
$\qquad\quad \{A[a \mathinner{..} i] \leqslant A[b] \leqslant A[j \mathinner{..} b]\};$
$\qquad\quad \langle A[a \mathinner{..} b], i, j\rangle := \langle A'[a \mathinner{..} b], i', j'\rangle.$
$\qquad\qquad perm(A[a \mathinner{..} b], A'[a \mathinner{..} b]) \wedge (i' > i \vee j' < j) \wedge (A[a \mathinner{..} i'-1] \leqslant A[b] \leqslant A[j'+1 \mathinner{..} b])\ \underline{\mathbf{od}};$
$\quad \langle A[j+1], A[b]\rangle := \langle A[p], A[j+1]\rangle;\ p := j\ \underline{\mathbf{end}}$

To prove that this correctly refines the **join** above we can use the theorem on termination of recursion (Theorem 1) to prove that the **while** loop terminates (first using the definitional transformation of **while** to express it as a suitable tail recursion). The term $\max(j-i, -1)$ is reduced at each execution of the body and always $\geqslant -1$ and the body of the loop must terminate. The invariant plus termination condition together prove that the loop establishes $A[a \mathinner{..} p-1] \leqslant A[p] \leqslant A[p+1 \mathinner{..} b]$. Finally, note that the body of the loop is a refinement of: $\langle A[a \mathinner{..} b], i, j\rangle := \langle A'[a \mathinner{..} b], i', j'\rangle.perm(A[a \mathinner{..} b], A'[a \mathinner{..} b]) \wedge (i' > i \vee j' < j)$ and a loop with this body is a refinement of $PERM$. So by Theorem 3 this is a correct refinement of the **join**.

Finally, to refine the atomic description we consider the cases $A[i+1] \leqslant A[b]$, $A[j-1] \geqslant A[b]$. In these cases we can increment i or decrement j as appropriate. For the case $A[i+1] \geqslant A[b]$, $A[j-1] \leqslant A[b]$ we can swap $A[i+1]$ and $A[j-1]$ and increment i and decrement j. This leads to the following executable implementation of quicksort:

$$
\begin{array}{l}
\underline{\mathbf{proc}}\ QSORT(a,b)\ \equiv \\
\underline{\mathbf{if}}\ b > a \\
\quad \underline{\mathbf{then}}\ \underline{\mathbf{begin}}\ p: \\
\qquad\qquad \underline{\mathbf{begin}}\ i := a-1,\ j := b: \\
\qquad\qquad\quad \underline{\mathbf{while}}\ i < j-1\ \underline{\mathbf{do}} \\
\qquad\qquad\qquad\quad \underline{\mathbf{if}}\ A[i+1] \leqslant A[b] \rightarrow i := i+1 \\
\qquad\qquad\qquad\quad \square\ A[j-1] \geqslant A[b] \rightarrow j := j-1 \\
\qquad\qquad\qquad\quad \square\ A[i+1] \geqslant A[b]\ \wedge\ A[j-1] \leqslant A[b] \rightarrow i := i+1;\ j := j-1; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle\ \underline{\mathbf{fi}}\ \underline{\mathbf{od}}; \\
\qquad\qquad\quad \langle A[j], A[b]\rangle := \langle A[p], A[j]\rangle;\ p := j\ \underline{\mathbf{end}} \\
\qquad\qquad QSORT(a, p-1);\ QSORT(p+1, b)\ \underline{\mathbf{end}}\ \underline{\mathbf{fi}}.
\end{array}
$$

This algorithm, although proven correct, does have some flaws as far as efficiency is concerned. The algorithm degenerates to $O(n^2)$ in the case of an already sorted array. It also does many more tests in the inner loop than necessary. To ensure the best efficiency it is necessary to try to split the array into roughly equal-sized components at each step. Sedgewick [4] suggests using the median of three elements taken from the array as the pivot. The elements being the first, last and middle elements in the array. Naturally, this is only sensible when there are at least three elements in the array. With this choice the recursive program looks like this:

$$
\begin{array}{l}
\underline{\mathbf{proc}}\ QSORT(a,b)\ \equiv \\
\underline{\mathbf{if}}\ b - a < 4 \\
\quad \underline{\mathbf{then}}\ \{b - a < 4\};\ SORT(a,b) \\
\quad\ \underline{\mathbf{else}}\ \underline{\mathbf{begin}}\ p: \\
\qquad\qquad p := \lfloor (a+b)/2 \rfloor; \\
\qquad\qquad \underline{\mathbf{if}}\ A[a] > A[p]\ \underline{\mathbf{then}}\ \langle A[a], A[p]\rangle := \langle A[p], A[a]\rangle\ \underline{\mathbf{fi}}; \\
\qquad\qquad \underline{\mathbf{if}}\ A[a] > A[b]\ \underline{\mathbf{then}}\ \langle A[a], A[b]\rangle := \langle A[b], A[a]\rangle\ \underline{\mathbf{fi}}; \\
\qquad\qquad \underline{\mathbf{if}}\ A[p] > A[b]\ \underline{\mathbf{then}}\ \langle A[p], A[b]\rangle := \langle A[b], A[p]\rangle\ \underline{\mathbf{fi}}; \\
\qquad\qquad \langle A[p], A[b-1]\rangle := \langle A[b-1], A[p]\rangle; \\
\qquad\qquad \{A[a] \leqslant A[b-1] \leqslant A[b]\}; \\
\qquad\qquad \underline{\mathbf{join}}\ \langle A[a+1\,..\,b-2], p\rangle := \langle A'[a+1\,..\,b-2], p'\rangle. \\
\qquad\qquad\qquad\qquad perm(A[a+1\,..\,b-2], A'[a+1\,..\,b-2]) \\
\qquad\qquad\quad \sqcup\ \langle A[a\,..\,b], p\rangle/\langle\rangle.a+1 \leqslant p \leqslant b-2\ \wedge\ (A[a\,..\,p] \leqslant A[b-1] \leqslant A[p+1\,..\,b])\ \underline{\mathbf{nioj}}; \\
\qquad\qquad \langle A[p], A[b-1]\rangle := \langle A[b-1], A[p]\rangle; \\
\qquad\qquad QSORT(a, p-1);\ QSORT(p+1, b)\ \underline{\mathbf{end}}\ \underline{\mathbf{fi}}.
\end{array}
$$

The proof that this refines $SORT$ is straightforward. For the implementation of the **join** we replace two occurrences of p by new variables i and j and introduce a loop with the invariant: $A[a\,..\,i-1] \leqslant A[b-1] \leqslant A[j+1\,..\,b]$. We can initialise the invariant with the assignments $i := a$; $j := b-1$. We get the following loop which implements the **join** statement above:

$$
\begin{array}{l}
\{A[a] \leqslant A[b-1] \leqslant A[b]\};\ \underline{\mathbf{begin}}\ i := a+1;\ j := b-2: \\
\qquad\qquad\qquad\qquad\qquad \underline{\mathbf{while}}\ i \leqslant j\ \underline{\mathbf{do}} \\
\qquad\qquad\qquad\qquad\qquad\qquad \underline{\mathbf{if}}\ A[i] \leqslant A[b-1] \rightarrow i := i+1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \square\ A[j] \geqslant A[b-1] \rightarrow j := j-1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \square\ A[i] \geqslant A[b-1]\ \wedge\ A[j] \leqslant A[b-1] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1\ \underline{\mathbf{fi}}\ \underline{\mathbf{od}}; \\
\qquad\qquad\qquad\qquad\qquad p := j+1\ \underline{\mathbf{end}}
\end{array}
$$

The next stage is to use program transformations to turn this into an efficient algorithm. First we resolve the nondeterminacy in the **if** statement in such a way that the third arm is chosen in preference to the others (this ensures that the subarrays are more equal in size when there are many equal elements):

$\{A[a] \leqslant A[b-1] \leqslant A[b]\}$; **begin** $i := a+1;\ j := b-2$:
$\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
while $i \leqslant j$ **do**
if $A[i] < A[b-1]$ **then** $i := i+1$
elsif $A[j] > A[b-1]$ **then** $j := j-1$
else $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi** **od**;
$p := j+1$ **end**

Next we do an "entire loop unroll" which copies the loop into one terminal position:

begin $i := a+1;\ j := b-2$:
$\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
while $i \leqslant j$ **do**
if $A[i] < A[b-1]$ **then** $i := i+1$;
while $i \leqslant j \ \wedge\ A[i] < A[b-1]$ **do** $i := i+1$ **od**
elsif $A[j] > A[b-1]$ **then** $j := j-1$
else $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi** **od**;
$p := j+1$ **end**

For the inner **while** loop we know that $A[i+1] < A[b-1] \Rightarrow i \leqslant j$ because from the loop invariant $A[j] \geqslant A[b-1]$. So we can remove this test from the **while**. We can insert a similar loop in the second arm of the **if** statement to get:

begin $i := a+1;\ j := b-2$:
$\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
while $i \leqslant j$ **do**
if $A[i] < A[b-1]$ **then** $i := i+1$;
while $A[i] < A[b-1]$ **do** $i := i+1$ **od**
elsif $A[j] > A[b-1]$ **then** $j := j-1$
while $A[j] > A[b-1]$ **do** $j := j-1$ **od**
else $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi** **od**;
$p := j+1$ **end**

After the first inner **while** loop we have $A[i] \geqslant A[b-1]$. If we selectively unroll the body of the loop with the extra test $A[j] > A[b-1]$ then we can prune it as follows:

begin $i := a+1;\ j := b-2$:
$\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
while $i \leqslant j$ **do**
if $A[i] < A[b-1]$
then $i := i+1$;
while $A[i] < A[b-1]$ **do** $i := i+1$ **od**;
if $A[j] > A[b-1]$ **then** $j := j-1$
while $A[j] > A[b-1]$ **do** $j := j-1$ **od** $\quad *$
else $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi**
elsif $A[j] > A[b-1]$
then $j := j-1$
while $A[j] > A[b-1]$ **do** $j := j-1$ **od**
else $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi** **od**;
$p := j+1$ **end**

At the point $*$ we have $A[i] \geqslant A[b-1]$ and $A[j] \leqslant A[b-1]$ so if we unroll here the body reduces to **if** $i \leqslant j$ **then** $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi**. With some simple manipulation, the body of the first **if** clause becomes:

while $A[i] < A[b-1]$ **do** $i := i+1$ **od**;
while $A[j] > A[b-1]$ **do** $j := j-1$ **od**;
if $i \leqslant j$ **then** $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi**

We can similarly express the second **if** clause as:

while $A[j] > A[b-1]$ **do** $j := j-1$ **od**;
if $i \leqslant j$ **then** $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi**

and insert **while** $A[i] < A[b-1]$ **do** $i := i+1$ **od** at the beginning. We can do the same with the third **if** clause (insert two redundant **while** loops and a redundant test). With these additions the whole **if** test is redundant so the loop can be simplified to:

begin $i := a+1;\ j := b-2$:
  $\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
  while $i \leqslant j$ **do**
    while $A[i] < A[b-1]$ **do** $i := i+1$ **od**;
    while $A[j] > A[b-1]$ **do** $j := j-1$ **od**;
    if $i \leqslant j$ **then** $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **fi** **od**;
  $p := j+1$ **end**

Re-write the loops as **do** ... **od** loops:

begin $i := a+1;\ j := b-2$:
  $\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
  do **if** $i > j$ **then** **exit** **fi**;
    while $A[i] < A[b-1]$ **do** $i := i+1$ **od**;
    while $A[j] > A[b-1]$ **do** $j := j-1$ **od**;
    if $i > j$ **then** **exit** **fi**;
    $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle;\ i := i+1;\ j := j-1$ **od**;
  $p := j+1$ **end**

Now the condition $i \leqslant j$ is true initially and true at the end of the loop body, so the first test for **exit** can be removed. The statements $i := i+1;\ j := j-1$ can be moved to the beginning of the loop by initialising to $i := a;\ j := b-1$. These can be absorbed into the inner **while** loops by "loop inversion":

begin $i := a;\ j := b-1$:
  $\{A[i] \leqslant A[b-1] \leqslant A[j]\}$;
  do **do** $i := i+1$;
      if $A[i] \geqslant A[b-1]$ **then** **exit** **fi** **od**;
    do $j := j-1$;
      if $A[j] \leqslant A[b-1]$ **then** **exit** **fi** **od**;
    if $i > j$ **then** **exit** **fi**;
    $\langle A[i], A[j]\rangle := \langle A[j], A[i]\rangle$ **od**;
  $p := j+1$ **end**

Here we have replaced **while** loops by loops with multiple **exit**s, the definitional transformations for these constructs are given in [6], their use is argued by Bochmann [1] and Taylor [5] among others.

This is a highly optimised version; the inner loops basically do a single test and increment. However, a simple insertion sort is still more efficient for "small" partitions. We can get the best of both worlds by using quicksort on the large partitions and ignoring small partitions. This results in a file which is "nearly sorted" and which can be sorted efficiently in linear time using insertion sort. To develop this algorithm replace the original *SORT* specification by the following *NSORT* (nearly sort):

$NSORT(a, b) \ =_{\mathrm{DF}}$ **join** $\langle A[a\,..\,b]\rangle / \langle\rangle.nsorted(A[a\,..\,b]) \sqcup PERM(a, b)$ **nioj** where

$$nsorted(A[a\,..\,b]) \;=_{\text{DF}}\; \forall a \leqslant i <= b.\, A[a\,..\,i-k] \leqslant A[i] \leqslant A[i+k\,..\,b]$$

where k is the size of the partitions to ignore (with $k=1$, $NSORT$ is equivalent to $SORT$). With this specification, for $b-a+1<k$, $NSORT$ is refined by **skip**. This leads to the following nearly-sorting algorithm:

```
proc NQSORT(a, b) ≡
if b − a + 1 < k
   then skip
    else begin p :
            p := ⌊(a + b)/2⌋;
            if A[a] > A[p] then ⟨A[a], A[p]⟩ := ⟨A[p], A[a]⟩ fi;
            if A[a] > A[b] then ⟨A[a], A[b]⟩ := ⟨A[b], A[a]⟩ fi;
            if A[p] > A[b] then ⟨A[p], A[b]⟩ := ⟨A[b], A[p]⟩ fi;
            ⟨A[p], A[b − 1]⟩ := ⟨A[b − 1], A[p]⟩;
            begin i := a; j := b − 1 :
               {A[i] ⩽ A[b − 1] ⩽ A[j]};
               do do i := i + 1;
                        if A[i] ⩾ A[b − 1] then exit fi od;
                  do j := j − 1;
                        if A[j] ⩽ A[b − 1] then exit fi od;
                  if i > j then exit fi;
                  ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩ od;
               p := j + 1 end
            ⟨A[p], A[b − 1]⟩ := ⟨A[b − 1], A[p]⟩;
            NQSORT(a, p − 1); NQSORT(p + 1, b) end fi.
```

Removing the recursion and introducing a stack AS to implement the parameters a and b we get the following algorithm:

```
proc QSORT(a, b) ≡ begin i, j, v, AS :
                       AS := ⟨⟩;
                       do if b − a + 1 > k
                            then j := ⌊(a + b)/2⌋;
                                 if A[a] > A[j] then ⟨A[a], A[j]⟩ := ⟨A[j], A[a]⟩ fi;
                                 if A[a] > A[b] then ⟨A[a], A[b]⟩ := ⟨A[b], A[a]⟩ fi;
                                 if A[j] > A[b] then ⟨A[j], A[b]⟩ := ⟨A[b], A[j]⟩ fi;
                                 ⟨A[j], A[b − 1]⟩ := ⟨A[b − 1], A[j]⟩;
                                 i := a; j := b − 1; v := A[b − 1];
                                 do do i := i + 1;
                                          if A[i] ⩾ j then exit fi od;
                                    do j := j − 1;
                                          if A[j] ⩽ j then exit fi od;
                                    if i > j then exit fi;
                                    ⟨A[i], A[j]⟩ := ⟨A[j], A[i]⟩ od;
                                 A[b − 1] := A[j + 1]; A[j + 1] := v;
                                 if j − a > b − j − 2
                                   then AS ← a; AS ← j; a := j + 2
                                    else AS ← j + 2; AS ← b; b := j fi
                             else if AS = ⟨⟩ then exit
                                              else b ← AS; a ← AS fi fi od;
                       for i := a + 1 to b step 1 do
                          v := a[i]; j := i;
```

$$
\begin{array}{l}
\underline{\textbf{while}}\ A[j-1] > v\ \underline{\textbf{do}} \\
\qquad A[j] := A[j-1];\ j := j-1\ \underline{\textbf{od}}; \\
A[j] := v\ \underline{\textbf{od}}\ \underline{\textbf{end}}.
\end{array}
$$

Here, the first loop uses nearly-quicksort to get A sorted to within k (a parameter to be empirically determined), then the **for** loop finishes the job using insertion sort.

Bibliography

[1] G. V. Bochmann, "Multiple exits from a loop without the goto," *Comm. ACM* 16 (July, 1973), 443–444.

[2] D. Gries, *The Science of Programming*, Springer-Verlag, New York–Heidelberg–Berlin, 1981.

[3] C. A. R. Hoare, "Quicksort," *Comput. J.* 5 (1962).

[4] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1988.

[5] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[6] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.