Chapter 10

# Case study: Insertion Sort

Insertion Sort is one of the simplest sorting algorithms, and will be our first case study involving sequences (or arrays).

The number of comparisons it makes is proportional on average to the square of the number of elements to be sorted. Later we will do better than that; but for now we study Insertion Sort as our first example of nested iterations.

## 10.1 What it means to be sorted

We are given sequence $as$ of integers, and we must rearrange its elements so that they are 'sorted'. To be more precise, we define a predicate 'is in non-strict ascending order',

$$\mathsf{up}\, as \ \widehat{=}\ (\forall\, i, j : 0{\rightarrow}\#as \cdot i \leq j \Rightarrow as[i] \leq as[j]) \ ,$$

by which we mean 'if the index of one element is no greater than the index of some other element, then the value of that element is no greater than the value of the other'.

With $\mathsf{up}$, we can start with the following abstract program:

$$\begin{array}{l} \textbf{var}\ as : \textbf{seq}_N\, \mathbb{Z};\ \ \textbf{con}\ A; \\ \textbf{and}\ A = \mathsf{bag}\, as \cdot \\ as\colon [\mathsf{up}\, as]\ . \end{array}$$

The variable $as$ is, of course, the sequence we are to sort.

The logical constant $A$ is the bag of elements in the sequence, and the invariant $A = \mathsf{bag}\, as$ means therefore that elements may neither be added to nor removed from $as$. Thus the sequence can be rearranged but not otherwise altered, and so we exclude trivial code such as $as := 0{\rightarrow}N$.

## 10.2 Similar pre- and postconditions

We approach the problem by successively sorting larger prefixes of *as*: at first, no matter what values *as* contains, still its empty prefix is sorted. What we want is to make its 'longest prefix' sorted, since the longest prefix of a sequence is the sequence itself.

The approach above is suggested to us, in fact, by the text of the abstract program, if we try to make its pre- and postconditions similar. By introducing prefixes explicitly,

$$\sqsubseteq \; as\colon [\mathsf{up}\; as{\uparrow}0 \;,\; \mathsf{up}\; as{\uparrow}N] \;\;,$$

we can see that somehow we want 'to change the 0 into an $N$'.

Both 0 and $N$ are constants, yet we want them to vary — therefore we replace them both by a new variable $k$, and vary *that*: variable $k$ can move from 0 to $N$, allowing us to write $\uparrow k$ in the pre- and postcondition. From there, the development of our first iteration is routine:

$$
\begin{array}{ll}
\sqsubseteq\; \mathbf{var}\; k : \mathbb{N}\bullet & \\
\quad k := 0; & \text{(i)} \\
\quad as, k\colon [k = 0 \;,\; \mathsf{up}\; as{\uparrow}k \;,\; k = N] & \triangleleft \\
\sqsubseteq\; I \;\widehat{=}\; k \le N \wedge \mathsf{up}\; as{\uparrow}k \bullet & \\
\quad as, k\colon [I \;,\; I \wedge k = N] & \\
\sqsubseteq\; \text{“invariant } I\text{, variant } N - k\text{”} & \\
\quad \mathbf{do}\; k \ne N \rightarrow & \\
\qquad as, k\colon [k < N \;,\; I \;,\; k > k_0] & \triangleleft \\
\quad \mathbf{od}\;. &
\end{array}
$$

(See Exercise 10.1 if you are puzzled about (i).)

Note how in the precondition above (writing $k < N$ rather than $k \ne N$) and in the postcondition (omitting $N \ge k$) we have made use of the invariant $I$.

## 10.3 Decreasing the variant

In many cases the easiest way is to deal with a variant is to decrease it explicitly, and we do that here with *following assignment* 3.5. After applying *specification invariant* 5.6, we proceed as follows:

$$
\begin{array}{lll}
& \sqsubseteq\; \text{“\textit{following assignment} 3.5”} & \\
& \quad as, k\colon [k < N \wedge I \;,\; I[k \backslash k+1] \wedge (k+1) > k_0]\,; & \triangleleft \\
& \quad k := k + 1 & \\
& \sqsubseteq\; \text{“\textit{contract frame} 5.4”} & \\
& \quad as\colon [k < N \wedge I \;,\; I[k \backslash k+1]] & \text{(ii)} \\
\text{(ii)} & \sqsubseteq\; as\colon [k < N \wedge \mathsf{up}\; as{\uparrow}k \;,\; \mathsf{up}\; as{\uparrow}(k+1)]\;. & \text{(iii)}
\end{array}
$$

Specification (ii) is an extremely common pattern in iteration bodies: assume truth of the invariant at $k$, then establish its truth at $k+1$.

## 10.4 Iterating up, and down

If the pattern so far is such a common one, then we really should see whether we can generalise it for later use. Suppose $N \geq 0$; then given a specification

$$as, k\colon [k = 0 \ , \ I \ , \ k = N] \ ,$$

we can by choosing invariant $I \wedge 0 \leq k \leq N$ develop the iteration

$$\begin{array}{ll}
\mathbf{do}\ k \neq N \rightarrow & \\
\quad as\colon [I \wedge 0 \leq k < N \ , \ I[k\backslash k+1]]\,; & \triangleleft \\
\quad k := k+1 & \\
\mathbf{od} &
\end{array}$$

by following steps like those in the previous section; the marked statement — where development continues — produces code to ensure that, when the subsequent $k := k+1$ is executed, the invariant will be re-established. Let us call the refinement above *iterate up*.

A second possibility for iterating up is to increase $k$ first, then re-establish the invariant. Then we have instead

$$\begin{array}{l}
\mathbf{do}\ k \neq N \rightarrow \\
\quad k := k+1; \\
\quad as\colon [I[k\backslash k-1] \wedge 0 < k \leq N \ , \ I] \\
\mathbf{od}\ .
\end{array}$$

Similar possibilities, for decreasing $k$ (from $N$ to 0 rather than from 0 to $N$), are examined in Exercise 10.9, where we call them *iterating down*.

With our 'packaged' up-iteration, we could redo our development so far as follows:

$$\begin{array}{lll}
 & \mathbf{var}\ as : \mathbf{seq}_N\, \mathbb{Z};\ \ \mathbf{con}\ A; & \\
 & \mathbf{and}\ A = \mathsf{bag}\, as\,\bullet & \\
 & as\colon [\mathsf{up}\, as] & \triangleleft \\
\sqsubseteq & \mathbf{var}\ k : \mathbb{N}\bullet & \\
 & k := 0; & \\
 & as, k\colon [k = 0 \ , \ \mathsf{up}\, as{\uparrow}k \ , \ k = N] & \triangleleft \\
\sqsubseteq & \text{“iterate up”} & \\
 & \mathbf{do}\ k \neq N \rightarrow & \\
 & \quad as\colon [k < N \wedge \mathsf{up}\, as{\uparrow}k \ , \ \mathsf{up}\, as{\uparrow}(k+1)]\,; & \text{(iii)} \\
 & \quad k := k+1 & \\
 & \mathbf{od}\ . &
\end{array}$$

## 10.5 A tricky invariant

With the iteration body (iii) we are left with what appears to be a fairly straightforward problem: given a sequence of length $k+1$ whose first $k$ elements are in order, effect a rearrangement that brings all *k+1* of its elements into order. (The 'sequence of length $k+1$' is the prefix $as{\uparrow}(k+1)$; in casting the problem as we have, we are taking a (small) chance by ignoring the possibility of using elements beyond the prefix (from $as{\downarrow}(k+1)$). But the chance we are taking is not that we will develop the *wrong* code — rather it is that we will develop no code at all.)

An obvious move for 'making the pre- and postconditions similar' is to write the postcondition as 'the first $k$ elements are sorted, and the last one is too':

$$\mathsf{up}\, as{\uparrow}(k+1) \quad \equiv \quad \mathsf{up}\, as{\uparrow}k \wedge as{\uparrow}k \le as[k] \ .$$

The resulting iteration would begin $\mathbf{do}\ \neg((as{\uparrow}k) \le as[k]) \rightarrow \cdots$, and the body would have to maintain $\mathsf{up}\, as{\uparrow}k$. Thinking about it operationally, however, there seems no way to move $as[k]$ *gradually* to its correct place in sequence $as$ while maintaining the order of $as{\uparrow}k$ — it would just have to be moved all at once, and then there would be no need for an iteration at all.

A slightly less obvious approach (but we are now forced to try a bit harder) is to generalise slightly: let the invariant be 'all but one of the elements are sorted' (so to speak — we will be more precise in a moment). Then initially the one element that is not is $as[k]$; finally it will be some other — say $as[l]$ — but we will add a conjunct to the postcondition 'element $as[l]$ is sorted as well'.

Now we must make that idea a bit more precise.

Again we try to make the precondition and postcondition similar. Since our concern is mainly with the prefix $as{\uparrow}(k+1)$, we call that $P$ (for 'prefix'), and the precondition can be written $\mathsf{up}\, P\backslash\{k\}$, meaning '$P$ is sorted except at $k$'. (We use upper case for $P$ to remind us that it is an expression, not a variable, and so cannot be assigned to.) In the postcondition we want, for some local variable $l$,

$$\mathsf{up}\, P\backslash\{l\} \wedge P{\uparrow}l \le P[l] \le P{\downarrow}(l+1) \ .$$

The first conjunct expresses that the prefix $P$ is sorted except at $l$; the second expresses that it is sorted at $l$ as well.

After all that preparation, we now have the refinement step, in which we add the local variable $l$. In the postcondition we constrain it to lie within the bounds of $P$ (it is non-negative anyway, because of its type):

$$\text{(iii)} \ \sqsubseteq \left\{ \begin{array}{lcll} P & \mathrel{\widehat{=}} & as{\uparrow}(k+1) & \text{an expression} \\ J & \mathrel{\widehat{=}} & P{\uparrow}l \le P[l] & \text{a formula} \\ K & \mathrel{\widehat{=}} & P[l] \le P{\downarrow}(l+1) & \text{a formula} \\ \mathbf{var}\ l : \mathbb{N} \bullet & & & \end{array} \right. .$$

$$as, l\colon [\mathsf{up}\, P\backslash\{k\}\ ,\ l \le k \wedge \mathsf{up}\, P\backslash\{l\} \wedge J \wedge K] \ .$$

| | |
|---|---|
| precondition | $\neg(P{\uparrow}l \le P[l])$<br>$l \le k$<br>$\mathsf{up}\, P\backslash\{l\}$<br>$P[l] \le P{\downarrow}(l+1)$ |
| postcondition | $l \ominus 1 \le k$<br>$\mathsf{up}\, P\backslash\{l \ominus 1\}$<br>$P[l \ominus 1] \le P{\downarrow}l$ |

**Figure 10.1** Pre- and postcondition of inner iteration body

Looking at the precondition, we see that $l$ should start at $k$, and so tend to 0. Looking at the postcondition, we see that there are two possibilities for the guard: either the negation of $J$ (leaving $K$ in the invariant), or the other way around. Since $l = 0 \Rrightarrow J$, we will take $\neg J$ as the guard, call the invariant $L$, and proceed:

$$
\begin{array}{ll}
\sqsubseteq\ L \ \widehat{=}\ \ l \le k \wedge \mathsf{up}\, P\backslash\{l\} \wedge K \bullet & \\
\quad l := k; & \\
\quad \textbf{do}\ \neg J \rightarrow & \\
\qquad as, l\colon [\neg J\ ,\ L\ ,\ l < l_0] & \triangleleft \\
\quad \textbf{od} & \\
\sqsubseteq\ as\colon [\neg J \wedge L\ ,\ L[l\backslash l \ominus 1]]\,; & \text{(iv)} \\
\quad l := l \ominus 1\ . &
\end{array}
$$

In (iv), we have met the pattern of (ii) again, this time decreasing. (It is not strictly speaking in the form of our simple 'iterate down', because the guard is not just $l \neq 0$. But they have many features in common.)

To make progress we must now, at last, expand our abbreviations to reveal in more detail what we have to work with: Figure 10.1 sets them out in tabular form. The main difference between the two is that the precondition is concerned with $l$, and the postcondition with $l \ominus 1$. To bring the two formulae closer together, we widen the sequence exclusions to $\{l \ominus 1, l\}$ in each case, adding back a conjunct to constrain $P[l \ominus 1]$ in the precondition, $P[l]$ in the postcondition. Doing that, and some simplification, results in Figure 10.2. (See Exercise 10.2.)

Comparing the four conjuncts one-by-one between pre- and postcondition, we find:

1. The first in the precondition implies the first in the postcondition.
2. The second conjuncts are complementary with respect to $l$ and $l \ominus 1$, suggesting a swap.
3. The third conjuncts are the same, even if we swap.

| | |
|---|---|
| precondition | $0 < l \leq k$<br>$P[l \ominus 1] > P[l]$<br>$\mathsf{up}\, P \backslash \{l \ominus 1, l\}$<br>$P{\uparrow}(l \ominus 1) \leq P[l \ominus 1] \leq P{\downarrow}(l+1)$ |
| postcondition | $0 < l \leq k+1$<br>$P[l \ominus 1] \leq P[l]$<br>$\mathsf{up}\, P \backslash \{l \ominus 1, l\}$<br>$P{\uparrow}(l \ominus 1) \leq P[l] \leq P{\downarrow}(l+1)$ |

**Figure 10.2** Rewriting of Figure 10.1.

4. The fourth conjunct in the precondition implies the fourth in the postcondition if we swap.

So 'swap' it is: we exchange elements $l \ominus 1$ and $l$ of $P$, and that is accomplished by exchanging them in $as$:

(iv) $\sqsubseteq$ *Swap* $(as, l \ominus 1, l)$ .

That leaves only the definition of *Swap* itself.

## 10.6 Assignment to sequences

In many programming languages, *Swap* $(as, l \ominus 1, l)$ would be written

$$as[l \ominus 1], as[l] := as[l], as[l \ominus 1] \ ,$$

in spite of there being *expressions* on the left of assignment. In general, what is meant by $as[i] := E$ is

$$as := as[i := E] \ ,$$

where $as[i := E]$ is the sequence got by replacing the $i^{th}$ element of $as$ by $E$. (Recall Section 9.5.3.) It is defined as follows:

*Abbreviation 10.1* sequence assignment For any sequence $as$, if $0 \leq i, j \leq \#as$ then

$$\begin{array}{lrll} as[i := E][j] \ \widehat{=} & E & \text{when} & i = j \\ & as[j] & \text{when} & i \neq j \ . \end{array}$$

□

Sequence assignment extends to multiple indices as in *Swap*: we have

$$\begin{array}{ll} & Swap\ (as, l \ominus 1, l) \\ = & as := as[l \ominus 1, l := as[l], as[l \ominus 1]] \ . \end{array} \qquad \text{(v)}$$

## 10.7 Removing the local invariant

All now is code except the local invariant $A = \mathsf{bag}\, as$. To remove it, we must check that every command in its scope maintains it. Since the only assignment made to $as$ is the one above, the swap, and clearly cannot violate the invariant $A = \mathsf{bag}\, as$, the local invariant can be removed, finally, leaving code.

## 10.8 Exercises

*Ex. 10.1* ♡ Explain the introduction of (i). What laws are used?

*Ex. 10.2* ♡ Recall Figure 10.2, rewritten from Figure 10.1. Why are the following true?

1. In the precondition, $P{\uparrow}l \not\leq P[l]$ can be replaced by $P[l \ominus 1] > P[l]$.
2. In the postcondition, $P[l \ominus 1] \leq P{\downarrow}l$ can be replaced by $P[l \ominus 1] \leq P[l]$.
3. In the precondition, $P[l] \leq P{\downarrow}(l+1)$ is not needed.

*Hint*: Use the other conjuncts too.

*Ex. 10.3* Use *sequence assignment* 10.1 and Figure 10.2 to check that (iv) $\sqsubseteq$ (v).

*Ex. 10.4* ♡ Replace the guard $\neg J$ with code. *Hint*: Use the invariant, and recall Exercise 10.2.

*Ex. 10.5* ♡ Here is the inner iteration of Insertion Sort:

$$
\begin{array}{l}
\mathbf{do}\ (l \neq 0) \wedge (as[l \ominus 1] > as[l]) \rightarrow \\
\qquad as := as[l \ominus 1, l := as[l], as[l \ominus 1]]; \\
\qquad l := l \ominus 1 \\
\mathbf{od}\ .
\end{array}
$$

Note that the value of $as[l]$ is the same on each iteration. We can 'tune' the algorithm to take advantage of that:

$$
\begin{array}{ll}
= & \mathbf{var}\ t : \mathbb{Z}\bullet \\
& t := as[l]; \\
& \mathbf{do}\ (l \neq 0) \wedge (as[l \ominus 1] > t) \rightarrow \\
& \qquad as := as[l := as[l \ominus 1]]; \\
& \qquad l := l \ominus 1 \\
& \mathbf{od}; \\
& as := as[l := t]\ .
\end{array}
$$

On each iteration we avoid two array index operations and one assignment.

Give the invariant for the tuned iteration. *Hint*: Consider the expression $P[l := t]$.

Why must the local invariant $A = \mathsf{bag}\, as$ have been removed *before* the program is changed as above?

*Ex. 10.6* ♡ Modify the predicate $\mathsf{up}$ so that it expresses *strict* order: moving from lower to higher indices must strictly increase the corresponding elements. (In other words, in the sorted sequence there may not be repeated elements.)

Why doesn't our development go through with the new definition? Find *precisely* the step that is in error.

*Ex. 10.7* *Binary search* You have a sorted sequence $as$ of integers, and must find the position $i$ of a given value $x$ in it. If $x$ is not in $as$, then $i$ should index the least element of $as$ greater than $x$, if any. Here is the abstract program:

$$\begin{array}{l} \textbf{var } as : \textbf{seq}_N\, \mathbb{Z};\ \ x : \mathbb{Z};\ \ i : \mathbb{N}; \\ \textbf{and } \mathsf{up}\, as\bullet \\ i\text{: } [as\uparrow i < x \le as\downarrow i] \ . \end{array}$$

Refine it to code; it should take time proportional to $\mathsf{log}\, N$ to execute. *Hint*: Recall Chapter 7.

*Ex. 10.8* As for assignment to sequences, for relation $r$ we allow an abbreviation for $r := r[s := t]$ — we just write

$$r[s] := t \ .$$

Similarly $r[ss] := t$ abbreviates $r := r[ss := t]$.

Now suppose that $g[s]$ equals some constant $c$ for all $s : S$ (hence $\mathsf{dom}\ g = S$). How could $f := f \oplus g$ be written in the style of $f[?] := ??$

What is the effect of $f[\{\}] := t$?

*Ex. 10.9* *Iterate down* Show that, provided $0 \le N$,

$$\begin{array}{ll} & as, k\text{: } [k = N\ ,\ I\ ,\ k = 0] \\ \sqsubseteq & \textbf{do } k \ne 0 \rightarrow \\ & \qquad as\text{: } [0 < k \le N \wedge I\ ,\ I[k \backslash k-1]]\,; \\ & \qquad k := k - 1 \\ & \textbf{od}\ . \end{array}$$

Where *precisely* is $0 \le N$ used?

*Ex. 10.10* ♡ Why aren't *iterate up* and *iterate down* made into laws?

*Ex. 10.11* ♡ In Exercise 10.9 the statement $k := k - 1$ appears in the body of the down iteration. Shouldn't it be $k := k \ominus 1$?