- Let us then turn to filling the row $1 \leq i \leq M$. Again we add a variable j for the next column to fill (omitting details):

```
$k := 0$; do $k \neq N+1 \to E[0][k]\,,\ k := k\,,\ k+1$ od;
$i := 1$;
{ outer invariant: Eq. (43)
  outer bound: number of still unfilled rows $M - i + 1$ }
do $i \neq M+1 \to$
    $j := 0$;
    { inner invariant:
            outer $\wedge\, \forall 0 \leq q < j : E[i][q] = \mathrm{ed}(i, q)$
      inner bound: still unfilled columns $N - j + 1$ }
    do $j \neq N+1 \to$
        fill entry $E[i][j]$;
        $j := j + 1$
    od;
    $i := i + 1$
od
```

- Note that this initialization $j := 0$ makes this addition to the outer invariant TRUE, so this inner invariant does indeed hold in the beginning of the inner loop.

- Note also that at the end of the inner loop we have inner invariant $\wedge$ ¬inner guard which means that row i has now been filled.

  This in turn shows that the outer invariant does indeed continue to hold even after incrementing i.

- However, column 0 is filled differently than the other columns > 0: by Eq. (40) instead of Eq. (41). Let us therefore separate thes two cases by rewriting the inner invariant into

$$\text{outer } \wedge (E[i][0] = \overbrace{\mathrm{ed}(i,0)}^{=i}) \wedge \forall 1 \leq q < j : E[i][q] = \mathrm{ed}(i,q)$$

  to see how this can be handled:

  initialization becomes $E[i][0], j := i, 1$

  filling entry $E[i][j]$ can now be done according to only Eq. (41).

```
$k := 0$; do $k \neq N+1 \to E[0][k]\,,\ k := k\,,\ k+1$ od;
$i := 1$;
{ outer invariant: Eq. (43)
  outer bound: number of still unfilled rows $M - i + 1$ }
do $i \neq M+1 \to$
    $E[i][0]\,,\ j := i\,,\ 1$;
    { inner invariant:
            outer $\wedge\, \forall 0 \leq q < j : E[i][q] = \mathrm{ed}(i, q)$
      inner bound: still unfilled columns $N - j + 1$ }
    do $j \neq N+1 \to$
        $t := \min(E[i][j-1], E[i-1][j]) + 1$;
        if $A[i] = B[j] \to$
```

```
                $t\ :=\ \min(t, E[i-1][j-1])$
            **fi** ;
            $E[i][j]\ :=\ t$ ;
            $j\ :=\ j+1$
        **od** ;
        $i\ :=\ i+1$
    **od**
```

Nested or Flat?

- We can write the edit distance algorithm (Section 3.4) also as one unnested loop:

```
$k\ :=\ 0$ ; **do** $k \neq N+1 \rightarrow E[0][k]\ ,\ k\ :=\ k\ ,\ k+1$ **od** ;
$i\ ,\ j\ :=\ 0\ ,\ N+1$ ;
{ joint invariant: Eq. (43) $\wedge \forall 0 \leq q < j : E[i][q] = \mathrm{ed}(i,q)$
  joint bound: number of still unfilled elements
      $(M-i) \cdot N + (N-j+1)$ }
**do** $(i \neq M+1) \wedge (j \neq N+1) \rightarrow$
     $t\ :=\ \min(E[i][j-1], E[i-1][j]) + 1$ ;
     **if** $A[i] = B[j] \rightarrow$
         $t\ :=\ \min(t, E[i-1][j-1])$
     **fi** ;
     $E[i][j]\ :=\ t$ ;
     $j\ :=\ j+1$
  $[\!]$ $(i \neq M) \wedge (j = N+1) \rightarrow$
     $i\ ,\ j\ ,\ E[i+1][0]\ :=\ i+1\ ,\ 1\ ,\ i+1$
**od**
```

- Note the guard of the second branch:
  - It says "if we have passed over the right end ($j = N+1$) of a line which is *not* the last ($i \neq M$), then. . . "
  - It comes from asking "under what conditions are we allowed to move to the next line ($i := i+1$) *and* initialize its first element ($E[i+1][0] := i+1$)?"
  - The first guard applies to the elements $1 \leq j \leq N$ of all rows, including the last.
  - The initialization of i and j state that row 0 has been filled (by the k loop).
- We prefer our loops to be

  nested if the inner loop(s) "run into completion".

  Here each row is fully filled by the inner loop by repeating it for the whole range $j := 1, 2, 3, \ldots, N$.

  unnested if it is necessary to break out of the inner loop in an orderly manner.

  This happens e.g. in our matrix search example (Figure 12) where we exit with success as soon as we find the desired x within out matrix b.

4 Sorting

The material in this section is from Dromey (1989, Chapter 9).

- Let us now develop several familiar sorting algorithms as further examples of programming guided by logic.
- The proofs of the different algorithms also show their common underlying ideas no longer visible in their codes.
- These ideas identify the following families:

  Insertion sorting proceeds by just taking the next element and inserting it into its correct place among the already sorted ones.

  Selection sorting finds instead a next element whose insertion is easy. Also *bubble-* and *heapsort* are selection sorting.

  Partition sorting splits instead its input into "small" and "large", sorts each separately, and combines their results.

  Quicksort splits cleverly (doable in many ways) to make combining simple.

  Mergesort does the opposite.

  We discuss them only after GCL *recursion*.

4.1 Counting and Permuting

- We have already seen a version of *insertion* sorting briefly (Section 3.3.4). There we noted that its invariant needed also a part of its precondition, and hinted that this occurs e.g. in just sorting.
- Informally, the postcondition of the sorting problem is that "the output array a is an *ordered permutation* of the input array A". We assume A to be nonempty.

  Ordered means that the elements of a are in nondescending order:

  $$\forall\, \mathrm{lower}(a) \le k < \mathrm{upper}(a) : a[k] \le a[k+1]. \tag{44}$$

  Clearly we would not call a "ordered" otherwise.

  Permutation means that the elements of a are the elements of A in some (usually) other order.

  Clearly we would not consider a as a result of "sorting" A if a *omitted* some elements, *invented* new elements on its own, or *duplicated* existing elements.

- However, stating "array b is a permutation of array d" as a logical formula — which we will denote as $perm(b, d)$ for brevity — is difficult, unless we add a new tool into out logical language.
- This new tool is the *counting quantifier:*

  $$\#x : \phi$$

  means "the number of such values for the variable x that the formula ϕ is TRUE".

- The other familiar quantifications $\forall y : \psi$ and $\exists z : \varphi$ turned a formula into another.

- In contrast, $\#x : \phi$ turns a formula into a natural number, which can then be used in expressions.

  (On the logic side $\{ \dots \}$, not in GCL code.)

- Then we can take as $\mathit{perm}(b, d)$ the formula

$$(\forall x : (\#\,\mathrm{lower}(b) \leq r \leq \mathrm{upper}(b) : b[r] = x) = \underbrace{(\#\,\mathrm{lower}(d) \leq s \leq \mathrm{upper}(d) : d[s] = x)}_{\text{the number of times } x \text{ appears within } d})$$

  or "for every possible element x, the number of times x appears within b = the number of times x appears within d".

- Then $\mathit{perm}(a, A)$ guarantees that no elements x are omitted, invented or duplicated.

- The initial sorting postcondition is therefore

$$\text{Formula (44)} \wedge \mathit{perm}(a, A). \tag{45}$$

  The only assignment it suggests is $a := A$, which does not get us started by itself. Hence we transform this postcondition into a programmer-friendler form. These transformations give the families.

- We take *swapping* two entries in a as our basic reoredering operation.

- The (trivial) basic result we need is that a is still a permutation of A even after a swap:

```
{ $\mathit{perm}(a, A)$ (and indices $i$ and $j$ are valid) }
$a[i]$ , $a[j]$ := $a[j]$ , $a[i]$ ;
{ $\mathit{perm}(a, A)$ }
```

- This in turn follows from our earlier verification of swapping two array elements (Section 2.4.8) once we verify as well that no other elements are affected:

$$\begin{aligned}
&\mathrm{wp}((a[i], a[j] := a[j], a[i]),\\
&\qquad \forall\,\mathrm{lower}(a) \leq k \leq \mathrm{upper}(a) : k \neq i \wedge k \neq j \Longrightarrow a[k] = B[k]\\
=&\forall\,\mathrm{lower}(a) \leq k \leq \mathrm{upper}(a) :\\
&\qquad k \neq i \wedge k \neq j \Longrightarrow \underbrace{\langle a; [i] \leftarrow a[j]; [j] \leftarrow a[i]\rangle\,[k]}_{= a[k] \text{ since } k \text{ is neither } i \text{ nor } j} = B[k].
\end{aligned}$$

  Hence we get the desired condition that a indeed remains similar to B for the other indices than i and j even after the swap.

4.2 The Common Specification

- Let us now strengthen the initial postcondition (45) into a common form for the three families.

- We omit mentioning the conjunct $\ldots \wedge \mathit{perm}(a, A)$ since it appears in all our formulas.

1. Let us begin by introducing a new variable q for the $k+1$ in Formula (44) (Section 3.3.3):

   $$\forall \operatorname{lower}(a) \leq k < \operatorname{upper}(a) : (q = k+1) \implies (a[k] \leq a[q]).$$

   - Here its definition $(q = k+1)$ becomes the antecedent of '$\implies$' because it is introduced within the scope of '$\forall$'.
   - Putting '$\wedge$' instead of '$\implies$' would have been wrong, since the meaning would be quite different!
   - This is a strengthening, as desired:

     $$\text{this step 1} \Rightarrow \text{the initial postcondition (45).}$$

2. Let us then introduce a quantifier over this q:

   $$\forall \operatorname{lower}(a) \leq k < \operatorname{upper}(a) : \\ \forall q : (k+1 \leq q) \wedge (q \leq \operatorname{upper}(a)) \implies (a[k] \leq a[q]).$$

   - We have not used this kind of strengthening before. How can we justify it?
   - We have seen before that '$\forall$' is an infinite '$\wedge$' when we justified shrinking its range as a valid weakening (Section 3.3.1).
   - Here we conversely *extend* its range: we go on from saying "$a[k] \leq a[q]$ when $q = k+1$" into saying "$a[k] \leq a[q]$ when $q = k+1$ *and* $a[k] \leq a[q]$ when $q = k+2$ *and* $a[k] \leq a[q]$ when $q = k+3$ *and* ... *and* $a[k] \leq a[q]$ when $q = \operatorname{upper}(a)$".
   - Hence we are indeed saying something stronger than before, and so we do indeed have

     $$\begin{aligned}\text{step 2} &\Rightarrow \text{step 1} \\ &\Rightarrow \text{the initial postcondition (45).}\end{aligned}$$

3. Let us continue by introducing another new variable p for k as in step 1:

   $$\forall \operatorname{lower}(a) \leq k < \operatorname{upper}(a) : (p = k) \implies \\ \forall q : (k+1 \leq q) \wedge (q \leq \operatorname{upper}(a)) \implies (a[p] \leq a[q]).$$

   Again we have

   $$\begin{aligned}\text{step 3} &\Rightarrow \text{step 2} \\ &\Rightarrow \text{the initial postcondition (45).}\end{aligned}$$

4. Let us finally add a quantifier over this p as in step 2:

   $$\forall \operatorname{lower}(a) \leq k < \operatorname{upper}(a) : \forall p : (\operatorname{lower}(a) \leq p) \wedge (p \leq k) \implies \\ \forall q : (k+1 \leq q) \wedge (q \leq \operatorname{upper}(a)) \implies (a[p] \leq a[q]).$$

   - We can also express these three quantifiers enclosing each other and their ranges as

     $$\forall \operatorname{lower}(a) \leq k < \operatorname{upper}(a), \operatorname{lower}(a) \leq p \leq k, \\ k+1 \leq q \leq \operatorname{upper}(a) : a[p] \leq a[q]. \quad (46)$$

   - This reads "for every position k, the elements $a[p]$ on the left side of k are at most as large as the elements $a[q]$ to the right side of k". (Figure 14)
   - We take this Formula (46) as our final definition of what "array a is ordered" means.

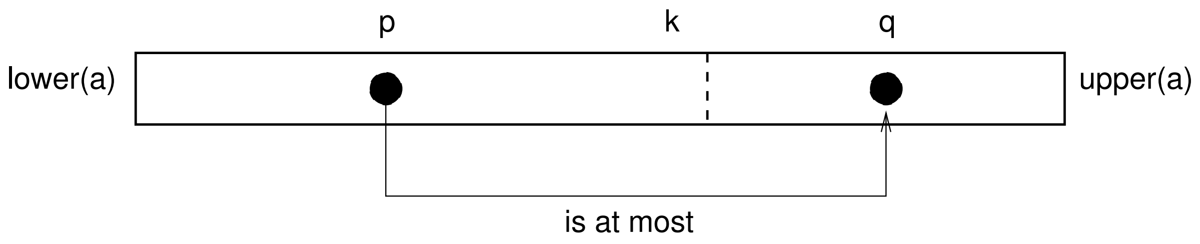

Figure 14: Sortedness Definition.

4.3 Array Sections

- Formula (46) and others like it can be written more explicitly, if we introduce notation for parts of arrays.
- This notation is *array section:*

  $$a[r \dots s]$$

  denotes a new array which consists of the contents of the array a restricted to the indices from r into s (inclusive).
- In the "arrays as functions" view (Section 2.4.8) its

  valid indices are $\mathrm{indices}(a) \cap \{r, r+1, r+2, \dots, s\}$

  value at such a valid index t is $a[t]$.
- We permit array sections on the right side of the GCL assignment command ':=' (Section 2.4.4).
  - This permits us to write e.g. $d := b[\mathrm{lower}(b) \dots \mathrm{lower}(b) + 7]$ to form a new array d consisting of the first 8 elements of b.
  - We do not permit array sections on the left side of ':='.
- On the logic side, we consider a subformula like

  $$a[r \dots s] \circledcirc \textit{something}$$

  as a convenient shorthand for the quantification

  $$\forall r \leq t \leq s : a[t] \circledcirc \textit{something}$$

  where t is some fresh variable name.
  - This says then that the property "$\dots \circledcirc$ *something*" holds for all elements of this section.
  - We consider also

    $$\textit{something} \circledcirc a[r \dots s]$$

    similarly.
- With this notation, Formula (46) can be expressed as

  $$\underbrace{\forall\, \mathrm{lower}(a) \leq k < \mathrm{upper}(a) : a[\mathrm{lower}(a) \dots k] \leq a[k+1 \dots \mathrm{upper}(a)]}_{\text{We call this formula } \textit{ordered}(a).}.$$

4.4 Insertion Sorting

- Let us now derive insertion sorting from this postcondition $\mathit{ordered}(a)$ (and the silently omitted $\mathit{perm}(a, A)$).
- The derivation begins by introducing a new variable i for *both* occurrences of $\mathrm{upper}(a)$:

$$\overbrace{(\forall\, \mathrm{lower}(a) \le k < i : a[\mathrm{lower}(a) \ldots k] \le a[k+1 \ldots i])}^{\text{Note that this is } \mathit{ordered}(a[\mathrm{lower}(a)\ldots i]).} \wedge (i = \mathrm{upper}(a)). \quad (47)$$

- Then we allow this i to change:

$$\mathit{ordered}(\overbrace{a[\mathrm{lower}(a) \ldots i]}^{\text{the left part}}) \wedge \overbrace{(\mathrm{lower}(a) \le i) \wedge (i \le \mathrm{upper}(a))}^{\text{The loop will guarantee this range.}}. \quad (48)$$

  We take this left part as our outer loop invariant. So our goal is to *keep this left part ordered* while incrementing i (by 1). The guard will be $i \ne \mathrm{upper}(a)$ so that at the end this (48) turns into the (47) above.

- Our algorithm is so far

```
$i$ , $a$ := lower($A$) , $A$ ;
**do** $i \ne$ upper($a$) $\rightarrow$
    "Increment $i$ by 1 while keeping invariant (48) TRUE."
**od**
```

- We start developing its body by asking "Under what conditions can we increment i by 1 while keeping invariant (48) TRUE?"

$$\begin{aligned} & \mathrm{wp}(i := i+1, \text{invariant } (48)) \\ = {} & \mathit{ordered}(a[\mathrm{lower}(a) \ldots i+1]) \\ \Leftrightarrow {} & \underbrace{\mathit{ordered}(a[\mathrm{lower}(a) \ldots i])}_{\text{invariant (48) itself}} \wedge (a[\mathrm{lower}(a) \ldots i] \le a[i+1]). \end{aligned} \quad (49)$$

- This new condition must be ensured by the outer loop body before it can increment i.
- It suggests an inner loop, since it contains a(n implicit) quantifier.
- So we add another variable j into the desired outcome of this inner loop:

$$\mathit{ordered}(a[\mathrm{lower}(a) \ldots j]) \wedge (a[\mathrm{lower}(a) \ldots j] \le a[i+1]) \wedge (j = i). \quad (50)$$

- Then we allow it to change to find our inner loop invariant:

$$\mathit{ordered}(a[\mathrm{lower}(a) \ldots j]) \wedge (\overbrace{a[\mathrm{lower}(a) \ldots j] \le a[i+1]}^{\text{The property to maintain.}}). \quad (51)$$

- A good initialization is $j := \mathrm{lower}(a) - 1$ since it makes the array sections $a[\mathrm{lower}(a) \ldots j]$ empty.

- We get our final *direct* insertion sort algorithm as

```
$i\,,\ a\ :=\ \mathrm{lower}(A)\,,\ \ A\,;$
$\{$ invariant: (48)
  bound: $\mathrm{upper}(a) - i\ \}$
**do** $i \neq \mathrm{upper}(a) \rightarrow$
    $j\ :=\ \mathrm{lower}(a) - 1\,;$
    $\{$ invariant: (51)
      bound: $i - j\ \}$
    **do** $j \neq i \rightarrow$
        **if** $a[j+1] \leq a[i+1] \rightarrow$
            **skip**
         $[\!]$ $a[j+1] > a[i+1] \rightarrow$
            $a[j+1]\,,\ a[i+1]\ :=\ a[i+1]\,,\ a[j+1]$
        **fi** ;
        $j\ :=\ j+1$
    **od** ;
    $i\ :=\ i+1$
**od**
```

Indirect Insertion Sort

- The inner loop can also proceed in the opposite order: from higher to lower indices.
- Then it can stop as soon as it has found the place for $a[i+1]$ without having to go through all the already ordered left part again. So it is more efficient and hence more widely known.
- We call it "indirect" because this way to insert $a[i+1]$ is not the most obvious one, as its construction shows.
- Its development deviates at (sub)formula (49) by writing it instead into the form

$$\mathit{ordered}(a[\mathrm{lower}(a)\ldots i+1]).$$

- Then we add the inner loop index j into this form:

$$\mathit{ordered}(a[j\ldots i+1]) \wedge (j = \mathrm{lower}(a))$$

- Then we split this array section into two at j:

$$\mathit{ordered}(\underbrace{a[\mathrm{lower}(a)\ldots j-1]}_{\text{Need to go here too?}}) \wedge \mathit{ordered}(\underbrace{a[j\ldots i+1]}_{\text{Been here.}}) \wedge (j = \mathrm{lower}(a)).$$

- But this split is not yet right: we must also ensure $a[\mathrm{lower}(a)\ldots j-1] \leq a[j\ldots i+1]$ so that these two split parts can be glued back together:

$$\mathit{ordered}(a[\mathrm{lower}(a)\ldots j-1]) \wedge \mathit{ordered}(a[j\ldots i+1]) \\ \wedge ((j = \mathrm{lower}(a))\,\mathbf{cor}\,(a[j-1] \leq a[j])).$$

- Initializing $j := i+1$ makes the *ordered* conjuncts TRUE but not the glue part. Hence they become the invariant and it becomes the negated guard (Section 3.3.2).

- Continuing this design leads eventually into

```
$i\,,\ a\ :=\ \mathrm{lower}(a)\,,\ \ A\,;$
do  $i \neq \mathrm{upper}(a) \rightarrow$
    $j\ :=\ i+1\,;$
    do  $(j \neq \mathrm{lower}(a))\,\mathbf{cand}\,(a[j-1] > a[j]) \rightarrow$
        $a[j-1]\,,\ a[j]\,,\ j\ :=\ a[j]\,,\ a[j-1]\,,\ j-1$
    od;
    $i\ :=\ i+1$
od
```

4.5 Selection Sorting

- The derivation starts to deviate from (47) when just *the first* occurrence of upper(a) gets replaced by i, not both:

$$(\forall\,\mathrm{lower}(a) \leq k < i : a[\mathrm{lower}(a)\ldots k] \leq a[k+1\ldots\mathrm{upper}(a)]) \wedge \underbrace{(i = \mathrm{upper}(a))}_{\neg\mathrm{guard}}.$$

  We again remember this particular value i as the one which restores the desired postcondition (47).

- Then allowing this i to change leads into

$$(\forall\,\mathrm{lower}(a) \leq k < i : a[\mathrm{lower}(a)\ldots k] \leq a[k+1\ldots\mathrm{upper}(a)]) \tag{52}$$

  which we again take as our outer loop invariant.

  - It reads "a looks *ordered* if you select any $k < i$ but not (yet) in general".
  - Then a is really *ordered* once we get this desired $i = \mathrm{upper}(a)$.

- Hence we again ask "How can we increment i in the outer loop and keep its invariant true?"

$$\begin{aligned}
&\mathrm{wp}(i := i+1, \text{invariant (52)})\\
=&(\forall\,\mathrm{lower}(a) \leq k < i+1 : a[\mathrm{lower}(a)\ldots k] \leq a[k+1\ldots\mathrm{upper}(a)])\\
\Leftrightarrow&(\forall\,\mathrm{lower}(a) \leq k < i+1 : a[\mathrm{lower}(a)\ldots k] \leq a[k+1\ldots\mathrm{upper}(a)])\\
\Leftrightarrow&\overbrace{(\forall\,\mathrm{lower}(a) \leq k < i : a[\mathrm{lower}(a)\ldots k] \leq a[k+1\ldots\mathrm{upper}(a)])}^{\text{Our outer invariant (52) again.}}\\
&\qquad\wedge (a[i] \leq a[i+1\ldots\mathrm{upper}(a)])
\end{aligned}$$

- This new part is a natural candidate for the outcome of our inner loop, because it contains a quantifier and is not implied by our outer invariant (52).

- Hence our developing code skeleton is now:

```
$i\,,\ a\ :=\ \mathrm{lower}(a)\,,\ \ A\,;$
do  $i \neq \mathrm{upper}(a) \rightarrow$
    { What invariant?
      What bound? }
```

```
        do  What guard? $\rightarrow$
                What body?
        od ;
        { Our design goal: the new part must hold here! }
        $i$ := $i + 1$
    od
```

- One natural way forward is to add into the new part a new variable j to be incremented in our inner loop:

$$\underbrace{(a[i] \leq a[i+1 \ldots j])}_{\text{This invariant!}} \wedge \underbrace{(j = \mathrm{upper}(a))}_{\text{Drop to } \neg\text{guard!}}. \tag{53}$$

  This inner invariant suggests also a natural initialization $j := i$.

- Hence we also chose $\mathrm{upper}(a) - j$ as our inner bound.

- Then we can develop the inner loop body by asking "How can we increment j and keep our inner invariant true?"

$$\begin{aligned} &\mathrm{wp}(j := j+1, a[i] \leq a[i+1 \ldots j]) \\ =&(a[i] \leq a[i+1 \ldots j+1]) \\ \Leftrightarrow&\underbrace{(a[i] \leq a[i+1 \ldots j])}_{\text{Our inner invariant.}} \wedge \underbrace{(a[i] \leq a[j+1])}_{\text{Ensure this – how?}}. \end{aligned}$$

  Hence we need to add code to ensure $a[i] \leq a[j+1]$ in front of the increment. Since we must also maintain the tacit $perm(a, A)$, we can (only) swap them if necessary.

- Our final direct selection sort algorithm is therefore:

```
    $i$ , $a$ := lower($a$) , $A$ ;
    { Invariant: (52)
      Bound: upper($a$) $-$ $i$ }
    do  $i \neq$ upper($a$) $\rightarrow$
        $j$ := $i$ ;
        { Invariant: $a[i] \leq a[i+1 \ldots j]$
          Bound: upper($a$) $-$ $j$ }
        do  $j \neq$ upper($a$) $\rightarrow$
            if  $a[i] \leq a[j+1]$ $\rightarrow$
                    skip
             [] $a[i] > a[j+1]$ $\rightarrow$
                    $a[i]$ , $a[j+1]$ := $a[j+1]$ , $a[i]$
            fi ;
            $j$ := $j + 1$
        od ;
        { The new part holds here by construction. }
        $i$ := $i + 1$
    od
```

Indirect Selection Sort

- Again we have also a more widely known indirect version.
  - It is more efficient, since its inner loop does not swap.
  - Instead, it just finds the place m where the single swap per outer loop can occurr.
- The development of its inner loop development continues from (53) by adding another new index variable m for i:

$$\overbrace{(a[m] \leq a[i+1\ldots j])}^{\text{What } m \text{ shall mean.}} \land (j = \mathrm{upper}(a)) \land (m = i).$$

  We shall therefore keep m pointed to the (first) smallest element among the entries $a[i+1\ldots j]$ already processed by the inner loop.

- The construction of the inner loop is otherwise similar to the direct case above.
- We get eventually:

```
$i$ , $a$ := lower($a$) , $A$ ;
{ Invariant: (52)
  Bound: upper($a$) − $i$ }
do $i \neq$ upper($a$) →
    $j$ , $m$ := $i$ , $i$ ;
    { Invariant: the meaning of $m$
      Bound: upper($a$) − $j$ }
    do $j \neq$ upper($a$) →
        if $a[m] \leq a[j+1]$ →
            skip
        [] $a[m] > a[j+1]$ →
            $m$ := $j+1$
        fi ;
        $j$ := $j+1$
    od ;
    { The swap will ensure the new part. }
    $a[i]$ , $a[m]$ , $i$ := $a[m]$ , $a[i]$ , $i+1$
od
```

- This development shows how logic-based design handles “peephole” changes to an existing algorithm:
  1. Go back to the postcondition of the code to change. So it is a good habit to keep them (or at least the most important ones) as code comments for later use.
  2. Modify this postcondition to describe also the meaning of this change.
  3. Redo the construction using this modified postcondition. Now it goes through a different route, since the meaning of the change must also be taken into account. Hence the code will be different too, and will contain the change (done correctly).

- Sometimes another such change is suggested too: to keep also the contents of $a[m]$ in another new variable x.
  - This is straightforward to do with the 3-step method outlined above.
  - This suggestion would then allow comparing x with $a[j + 1]$ in the **if** clause, saving the cost of repeatedly indexing $a[m]$.
  - However, this "benefit" is doubtful today: the cost of (re)assigning x too whenever m changes might overrun these savings, especially if copying is costly for the element type of array a (e.g. if it involves calling expensive constructors).