Correctness

Last updated: March 18$^{\text{th}}$ 2019, at 1.05pm

Contents

1 Proving Programmes

I.e., testing programmes. Hopefully you have been doing this throughout the logbook exercises.

However, as an exercise, this week's code bundle includes the model answer from week 11 (topological sorts). Two additional classes have been provided:

- `TopologicalSortTest`, and
- `DepthFirstTest`.

The first of these contains a stub (the `testGraph` method) for a test function for a topological sort, the second contains an example of how this test could be used. Complete the implementation of `testGraph`, and then use it to test both the model answer implementation of depth first sorting, and to test your implementation, from week 11, of reference counting sorting.

2 Proving Programmes Correct

2.1 Square

The following algorithm, also presented in the lecture, will calculate n^2.

```
public static int square(int n) {
   int i = 0; square = 0; twoN = 0;

   while (i < n) {
      square = square + twoN + 1;
      twoN = twoN + 2;
      i++;
   }
   return square;
}
```

This algorithm is also included in this week's code bundle, with Java assertions added, documenting the proof from the lecture.

Try running the programme, with assertions enabled, and check that none of the assertions fail. Remember! the fact that the programme runs with none of the assertions failing is *not* a proof, in itself, that the programme is correct. The proof is the logical anlysis of the correctness of the assertions.

In IntelliJ you can pass the `-ea` flag to java by selecting "Run Configurations..." from the Run menu. In the pop-up box that appears select the "Arguments" tab, and enter "`-ea`" in the "VM arguments" field.

Try changing the programme and/or the assertions so that one or more of the assertions will fail, and then run the programme again and observe its behaviour.

Model question. An annotated version of this algorithm, more or less mirroring the proof provided in the lecture, but with more documentation elucidating and justifying the assertions, will be provided as a model answer.

2.2 Cube

Logbook question. The algorithm shown below will calculate n^3.

```
public static int cube(int n) {
   int i=0, cube = 0, threeNsq = 0, threeN = 0;

   while (i < n) {
      cube = cube + threeNsq + threeN + 1;
      threeNsq = threeNsq + 2*threeN + 3;
      threeN = threeN + 3;
      i++;
   }
   return cube;
}
```

Prove that the programme above is correct. This code is also provided in this week's code bundle. Add assertions to this code, documenting your proof. Try running your programme with assertions enabled. For the logbook you should submit the written proof by assertion, with elucidation and justification of your assertions, *not* the Java code with Java assertions.

2.3 Binary Search

Model question. This section, and the next (2.4), provide longer and more complex examples of proof by assertion. However, these proofs are incomplete. For these exercises, you should complete these proofs.

In this exercise the following components of the proof are missing:

- In section 2.3.2.1, assertions $\boxed{6}$, $\boxed{13}$, and $\boxed{17}$;
- in section 2.3.2.2, elucidations $\boxed{15}$, $\boxed{22}$, and $\boxed{25}$;
- and in section 2.3.2.3, justifications $\boxed{8}$ and $\boxed{30}$,

You should provide these missing components of the proof.

2.3.1 The Basic Algorithm

The method in section 2.3.1.1 implements a binary search. The first parameter, `array`, is a sorted array of `Comparable`s. The second parameter, `target`, is the value we are looking for, known as the *target value*. If the method finds the target value it will return the index of the position in the array where it was found. If the target value is not in the array the method will return -1.

2.3.1.1 Implementation

```
public <T extends Comparable<T>> int binarySearch(T[] array,T target) {
   int first = 0, last = array.length-1; // initially search in the whole array
   while (first <= last) {// still at least one element in search space
      // calculate median index, halfway between first and last indices...
      int median = (first+last)/2;
      // ...and get the value there
      int comparison = array[median].compareTo[target];
      if (comparison == 0) {// target value found
         return median;
      } else if (comparison > 0) {// median value is larger than target
         last = median-1; // search further in lower half of search space
      } else {// median value is smaller than target
         first = median+1; // search further in upper half of search space
      }
   }
   return -1; // ran out of search space without finding target
}
```

2.3.1.2 Explanation The algorithm in section 2.3.1.1 works as follows.

On each iteration the method is searching in a section of the array, known as the *search space*, and delimited by the indices `first` and `last`. The search space is always such that it is guaranteed that the target value will be contained within it, if it is in the array at all. Initially, the search space is the whole array, so `first` and `last` are the first and last index of the whole array (i.e. zero and `array.length-1`). To search, the method looks at the middle entry in the search space — the *median value*. If this is the target value the method returns its index. Otherwise, if the median value is larger than the target value the method will continue its search in the lower half of the search space, and if the median value is smaller than the target value the search continues in the upper half of the current workspace.

2.3.2 The Annotated Algorithm

Section 2.3.2.1 contains the same algorithm as is in section 2.3.1.1, but now partially annotated with assertions aiming to show that the method has the correct semantics — i.e. that if the target is in the array the method returns the target's index, and that it otherwise returns -1.

If you are unsure how to read the assertions in section 2.3.2.1 please see section 2.3.2.2. This contains explanations of the assertions which should make their meaning clearer.

The assertions in section 2.3.2.1 are only presented with short comments explaining their derivation. Section 2.3.2.3 contains more detailed justifications for the assertions' derivations.

2.3.2.1 Assertions

```
public <T extends Comparable<T>> int binarySearch(T[] a,T t) {
    [1] {∀i : array[i] = target ⇒ 0 ≤ i ≤ array.length − 1}  // from properties of arrays
    [2] {∀i, j ∈ [0, array.length − 1] : i ≤ j ⇒ array[i] ≤ array[j]}  // given
    int first = 0, last = array.length-1; // initially search in the whole array
    [3] {first = 0, last = array.length − 1}  // from assignment above
    [4] {(∃i : array[i] = target) ⇒ first ≤ i ≤ last}  // from [1] and [3]
    while (first <= last) {// still at least one element in search space
        [5] {first ≤ last}  // from while test
        [6] { To be provided}  // from [4] and [26]
        int median = (first+last)/2; // median index is halfway between top and bottom
        [7] {median = (first+last)/2}  // from assignment above
        [8] {first ≤ median ≤ last}  // to be justified
        int comparison = array[median].compareTo(target);
        [9] {comparison = 0 ⇒ array[median] = target}  // from properies of compareTo
        [10] {comparison > 0 ⇒ array[median] > target}  // from properies of compareTo
        [11] {comparison < 0 ⇒ array[median] < target}  // from properies of compareTo
        if (comparison == 0) {// target value found
            [12] {comparison = 0}  // from if test
            [13] { To be provided}  // from [12] and [9]
            return median;
        }  else if (comparison > 0) {// median value is larger than target
            [14] {comparison > 0}  // from if test
            [15] {array[median] > target}  // from [14] and [10]
            [16] {∀n > median : array[n] > target}  // from [2] and [15]
            [17] { To be provided}  // from [6], [8], and [16]
            last = median-1; // search further in lower half of search space
            [18] {last = median − 1}  // from assignment above
            [19] {∃i : array[i] = target ⇒ first ≤ i ≤ last}  // from [17] and [18]
        }  else {// median value is smaller than target
            [20] {comparison < 0}  // from failure of two previous if tests
            [21] {array[median] < target}  // from [20] and [11]
            [22] {∀n < median : array[n] < target}  // from [2] and [21]
            [23] {(∃i : array[i] = target) ⇒ median + 1 ≤ i ≤ last}  // from [6], [8] and [23]
            first = median+1; // search further in upper half of search space
            [24] {first = median + 1}  // from assignment above
            [25] {(∃i : array[i] = target) ⇒ first ≤ i ≤ last}  // from [24] and [23]
        }
        [26] {(∃i : array[i] = target) ⇒ first ≤ i ≤ last}  // from [19] and [25]
    }
    [27] {last < first}  // from failue of while test
    [28] {∄n : first ≤ n ≤ last}  // from [27]
```

```
    [29]{∃i : array[i] = target ⇒ first ≤ i ≤ last} // from [4] and [26]
    [30]{∄n : array[n] = target} // to be justified
    return -1; // run out of search space without finding target
}
```

2.3.2.2 Elucidations The following elucidations should help to make the meanings of the assertions clearer.

$\boxed{1}$ If `target` is in the array its index will be somewhere between the first index in the array, 0, and the last index, `array.length` − `1`.

$\boxed{2}$ If `i` and `j` are valid indices in the array, and `i` is less than or equal to `j`, then the element at index `i`, i.e. `array`[`i`], will be less than `array`[`j`], the element at index `j`.

$\boxed{3}$ `first` is zero, `last` is `array.length` − `1`

$\boxed{4}$ If the target value is in the array its index will be somewhere between `first` and `last` (inclusive).

$\boxed{5}$ `first` comes before `last`

$\boxed{6}$ If the target value is in the array its index will be somewhere between `first` and `median` − `1` (inclusive).

$\boxed{7}$ `median` is the average of `first` and `last`.

$\boxed{8}$ `median` is between `first` and `last`, and `first` is less than or equal to `median`, and `median` is less than or equal to `last`.

$\boxed{9}$ If `comparison` is zero, then the array entry at `median` and the target are equal.

$\boxed{10}$ If `comparison` is greater than zero, then `array`[`median`] is greater than `target`.

$\boxed{11}$ If `comparison` is less than zero, then `array`[`median`] is less than `target`.

$\boxed{12}$ `comparison` is zero.

$\boxed{13}$ `array`[`median`] is equal to the target.

$\boxed{14}$ `comparison` is greater than zero.

$\boxed{15}$ *To be provided*

$\boxed{16}$ For any index greater than `median` the array entry will be greater than `target`.

$\boxed{17}$ If the target value is in the array its index will be somewhere between `first` and `median` $-$ `1` (inclusive).

$\boxed{18}$ `last` is now `median` $-$ `1`

$\boxed{19}$ If the target value is in the array its index will be somewhere between `first` and `last` (inclusive).

$\boxed{20}$ `comparison` is less than zero.

$\boxed{21}$ `array`[`median`] is less than the target.

$\boxed{22}$ *To be provided*

$\boxed{23}$ If the target value is in the array its index will be somewhere between `median` $+$ `1` and `last` (inclusive).

$\boxed{24}$ `first` is now `median` $+$ `1`

$\boxed{25}$ *To be provided*

$\boxed{26}$ If the target value is in the array its index will be somewhere between `first` and `last` (inclusive).

$\boxed{27}$ `last` is less than `first`.

$\boxed{28}$ There is no number `n` greater than or equal to `first` and less than or equal to `last`.

$\boxed{29}$ If the target value is in the array its index will be somewhere between `first` and `last` (inclusive).

$\boxed{30}$ The target value is not in the array — i.e. there is no index in the array at which the target value can be found.

2.3.2.3 Justifications The following are longer justifications for the derivation of the assertions.

$\boxed{1}$ This follows from the properties of arrays. If `target` is in the array, it must have a valid index, which must therefore lie between zero and `length` $-$ `1` inclusive.

$\boxed{2}$ Because the array is sorted, which is given.

$\boxed{3}$ From the assignment on line `4`. This assertion is also a way of stating that the search space encompasses the whole array.

$\boxed{4}$ From $\boxed{1}$ we have that the index must be between zero and `array.length` $-$ `1`, and from $\boxed{3}$ we have `first` $=$ `0` and `last` $=$ `array.length` $-$ `1`. Simply substituting `first` for `0` and `last` for `array.length` $-$ `1` in $\boxed{1}$ gives $\boxed{4}$.

$\boxed{5}$ Follows from the success of the **while** test.

$\boxed{6}$ This holds (from $\boxed{4}$) just before entering the **while** loop, and (from $\boxed{26}$) at the end of the **while** loop (just before looping round again). It must therefore also hold here.

$\boxed{7}$ From the assignment on line 10.

$\boxed{8}$ *To be provided*

$\boxed{9}$ From the properties of the `compareTo` method.

$\boxed{10}$ From the properties of the `compareTo` method.

$\boxed{11}$ From the properties of the `compareTo` method.

$\boxed{12}$ This follows from the success of the **if** test on line 17.

$\boxed{13}$ From $\boxed{12}$, $\texttt{comparison} = 0$, so, from $\boxed{9}$ we can conclude that $\texttt{array}[\texttt{median}] = \texttt{target}$.

$\boxed{14}$ This follows from the success of the **if** test on line 21.

$\boxed{15}$ From $\boxed{14}$, $\texttt{comparison} > 0$, so, from $\boxed{10}$ we can conclude that $\texttt{array}[\texttt{median}] > \texttt{target}$.

$\boxed{16}$ From $\boxed{15}$ the entry at `median` is already greater than `target`. From $\boxed{2}$ the array is sorted, so any entry beyond index `median` must also be greater than `target`.

$\boxed{17}$ From $\boxed{6}$ any correct index has to be between `first` and `last`. From $\boxed{8}$ `median` is between `first` and `last`. From $\boxed{17}$ any element with an index greater than or equal to `median` must be greater than `target` (and therefore not equal to `target`), so the largest possible correct index is $\texttt{median} - 1$. The smallest is still `first`.

$\boxed{18}$ From the assignment on line 26.

$\boxed{19}$ Simple replacement of $\texttt{median} - 1$ in $\boxed{17}$ by `last`, from $\boxed{18}$.

$\boxed{20}$ This follows from the failure of the **if** tests on lines 17 and 21. The test at line 17 must have failed, so $\texttt{comparison} \neq 0$, and test at line 21 has failed, so $\texttt{comparison} \not> 0$, so it must be the case that $\texttt{comparison} < 0$.

$\boxed{21}$ From $\boxed{20}$, $\texttt{comparison} < 0$, so, from $\boxed{11}$ we can conclude that $\texttt{array}[\texttt{median}] < \texttt{target}$.

$\boxed{22}$ From $\boxed{21}$ the entry at `median` is already less than `target`. From $\boxed{2}$ the array is sorted, so any entry below index `median` must also be less than `target`.

$\boxed{23}$ From $\boxed{6}$ any correct index has to be between `first` and `last`. From $\boxed{8}$ `median` is between `first` and `last`. From $\boxed{22}$ any element with an index less than or equal to `median` must be less than `target` (and therefore not equal to `target`), so the smallest possible correct index is `median`+`1`. The largest is still `last`.

$\boxed{24}$ From the assignment on line 34.

$\boxed{25}$ Simple replacement of `median` + `1` in $\boxed{23}$ by `first`, from $\boxed{24}$.

$\boxed{26}$ The only route to this point is through line 26, or through line 34. $\boxed{19}$ following line 26 and $\boxed{25}$ following line 34 are identical to this assertion.

$\boxed{27}$ To reach this point the **while** test (`first` $\leq$ `last`) must have failed, so `last` must be less than `first`.

$\boxed{28}$ This follows from $\boxed{27}$. Since `last` is less than `first`, there is no index `n` with `first` $\leq$ `n` and `n` $\leq$ `last`, as this would requite `first` $\leq$ `last`.

$\boxed{29}$ This is identical to $\boxed{4}$, which holds just before entering the **while** loop, and to $\boxed{26}$, which holds at the end of each iteration of the while loop. So it must also hold here.

$\boxed{30}$ *To be provided*

2.4 Bubble Sort

In this section the following components are missing from this proof of the correctness of the bubble sort algorithm:

- In section 2.4.2.2, assertion $\boxed{13}$ and $\boxed{19}$;
- in section 2.4.2.3, elucidations $\boxed{15}$ and $\boxed{34}$;
- and in section 2.4.2.4, justifications $\boxed{5}$, $\boxed{7}$, and $\boxed{14}$.

You should provide these missing components of the proof.

2.4.1 The Basic Algorithm

2.4.1.1 Implementation The method below implements bubble sort. The parameter, `array`, is an array of `Comparable`s. On completion of the method the array will be sorted in ascending order, i.e. with the smallest entry at index zero. The code assumes the existence of a `swap` method, which will swap the two values passed to it.

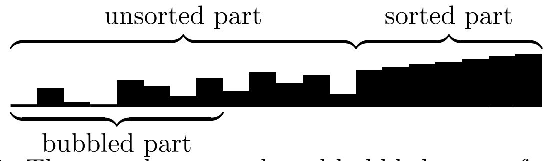

Figure 1: The sorted, unsorted, and bubbled parts of the array

```
1  public <T extends Comparable<T>> void sort(T[] array) {
2    for (int lastUnsorted = array.length-1; // the whole list is unsorted
3        lastUnsorted >= 0; // stop when the whole list is sorted
4        lastUnsorted--) {// one new element sorted each time round
5        for (int nextToCompare = 0; // start comparing the first two elements
6            nextToCompare < lastUnsorted; // stop at the end of the unsorted portion
7            nextToCompare++) {
8            if (array[nextToCompare].compareTo(array[nextToCompare+1]) > 0) {
9                // the elements are in the wrong order so swap them around
10               swap(array[nextToCompare],array[nextToCompare+1]);
11           }
12       }
13   }
14 }
15
```

2.4.1.2 Explanation The algorithm in section 2.4.1.1 will gradually sort the array from the end of the array to the start — i.e. from the highest index to the lowest. On each iteration of the outer **for** loop (starting at line 2) there will be an upper section of the array, the *sorted* part, that will be sorted — i.e. that will contain the correct entries in the correct places — and an *unsorted* lower part. In the initial iteration the sorted part is empty.

On each iteration of the inner **for** loop (starting at line 5) the method will pass through the unsorted part of the array comparing each entry to its neighbour. If the entries are in the wrong order — i.e. not increasing in size — they are swapped. The net effect is that at the end of the inner iteration the largest element in the unsorted part of the array has "bubbled" its way up to its correct position as last element in the unsorted part, and can therefore be added to the sorted part.

We can therefore also talk about a "bubbled" part of the array, in addition to the sorted and unsorted parts. The bubbled part will always be part of the unsorted part. At the end of each iteration of bubbling the bubbled part of the array covers the whole unsorted part. This is shown schematically in figure 1.

2.4.2 The Annotated Algorithm

2.4.2.1 Implementation The following is a variant of the code listed in section 2.4.1.1. This version uses **while** loops, rather than **for** loops, but otherwise works in the same way. The reason for switching from **for** loops to **while**

loops is that using **while** loops makes proving the code correct simpler.

```
public <T extends Comparable<T>> void sort(T[] array) {
  int lastUnsorted = array.length-1; // start sorting from the end of the array
  while (lastUnsorted >= 0) {// as long as the unsorted part is not empty
    int nextToCompare = 0; // start bubbling from the start of the array
    while (nextToCompare < lastUnsorted) {// bubble over the whole unsorted part
      if (array[nextToCompare].compareTo(array[nextToCompare+1]) > 0) {
        // the elements are in the wrong order so swap them around
        swap(array[nextToCompare],array[nextToCompare+1]);
      }
      nextToCompare = nextToCompare+1; // compare next pair
    }
    lastUnsorted = lastUnsorted-1; // one more element sorted
  }
}
```

Section 2.4.2.2 contains the same algorithm[1], but now partially annotated with assertions aiming to show that the method has the correct semantics — i.e. that on termination of the method the array will be sorted.

If you are unsure how to read the assertions in section 2.4.2.2 please see section 2.4.2.3. This contains explanations of the assertions which should make their meaning clearer.

The assertions in section 2.4.2.2 are only presented with short comments explaining their derivation. Section 2.4.2.4 contains more detailed justifications for the assertions' derivations.

In the assertions notations such as $\mathsf{i}, \mathsf{j} \in [0, \mathsf{n}]$ are used to restrict i and j to be in a specified range, so that, in this example, $\mathsf{i}, \mathsf{j} \in [0, \mathsf{n}]$ means that $0 \leq \mathsf{i} \leq \mathsf{n}$ and $0 \leq \mathsf{j} \leq \mathsf{n}$. In this context it is worth noting that the sorted part of the array will typically have indices in the range $[\mathsf{lastUnsorted} + 1, \mathsf{array.length} - 1]$, the unsorted part in the range $[0, \mathsf{lastUnsorted}]$, and the bubbled part in the range $[0, \mathsf{nextToCompare}]$.

In the assertions the following properties of `swap` are assumed:

1. $\mathtt{swap}(\mathtt{x}, \mathtt{y})$ will swap the values of `x` and `y`. I.e.

   $$\mathtt{swap}(\mathtt{x}, \mathtt{y});$$
   $$\{\mathsf{x}_{\mathsf{new}} = \mathsf{y}_{\mathsf{old}} \;\&\&\; \mathsf{y}_{\mathsf{new}} = \mathsf{x}_{\mathsf{old}}\}$$

   will be true. The subscripts $_{\mathsf{old}}$ and $_{\mathsf{new}}$ are used to refer to the values of the variables before and after the swap, respectively.

2. No values other than those in (1) will be changed.

[1]This is not quite true. The version in section 2.4.2.2 has an empty **else** part added to the **if** statement in order to allow some assertions to be added about what happens if the **if** test fails.

3. As a consequence of (2), if the two values to be swapped are members of the same array then after the call of `swap` the array will still contain the same values.

4. A stricter formulation of (2) allows us to consider just part of the array. If both values to be swapped are in a given section of the same array, then the swap will not change the set of values in that section of the array.

These properties will be useful in the proofs of some of the assertions in section 2.4.2.2.

2.4.2.2 Assertions

```
1   public <T extends Comparable<T>> void sort(T[] array) {
2     int lastUnsorted = array.length-1;
3     [1]{lastUnsorted = array.length − 1} // from assignment on line 2
4     [2]{∀i, j ∈ [lastUnsorted + 1, array.length − 1] : i < j ⇒ array[i] ≤ array[j]} // from [1]
5     [3]{∀i ∈ [0, lastUnsorted] : array[i] ≤ array[lastUnsorted + 1]} // from [1]
6     while (lastUnsorted >= 0) {
7       [4]{∀i, j ∈ [lastUnsorted + 1, array.length − 1] : i < j ⇒ array[i] ≤ array[j]}
8           // from [2], [30]
9       [5]{∀i ∈ [0, lastUnsorted] : array[i] ≤ array[lastUnsorted + 1]} // to be justified
10      int nextToCompare = 0;
11      [6]{nextToCompare = 0} // from assignment on line 10
12      [7]{∀i ∈ [0, nextToCompare − 1] : array[i] ≤ array[nextToCompare]} // to be justified
13      while (nextToCompare < lastUnsorted) {
14        [8]{nextToCompare < lastUnsorted} // from success of while test on line 13
15        [9]{∀i, j ∈ [lastUnsorted + 1, array.length − 1] : i < j ⇒ array[i] ≤ array[j]}
16            // from [4], [22]
17        [10]{∀i ∈ [0, lastUnsorted] : array[i] ≤ array[lastUnsorted + 1]} // from [5], [23]
18        [11]{∀i ∈ [0, nextToCompare − 1] : array[i] ≤ array[nextToCompare]}
19            // from [7], [21]
20        if (array[nextToCompare].compareTo(array[nextToCompare+1]) > 0) {
21          [12]{array_old[nextToCompare] > array_old[nextToCompare + 1]}
22              // from success of if test on line 20
23          swap(array[nextToCompare],array[nextToCompare+1]);
24          [13]{To be provided} // from property (1) of swap
25          [14]{members_new[0, lastUnsorted] = members_old[0, lastUnsorted]}
26              // to be justified
27          [15]{∀i ∈ [lastUnsorted + 1, array.length − 1] : array_new[i] = array_old[i]}
28              // from [8]
29          [16]{array_new[nextToCompare] < array_new[nextToCompare + 1]}
30              // from [12], [13]
31          [17]{∀i ∈ [0, nextToCompare] : array_new[i] ≤ array_new[nextToCompare + 1]}
```

32 // from $\boxed{11}$, $\boxed{16}$
33 } **else** {
34 $\boxed{18}\{\mathsf{array}[\mathsf{nextToCompare}] \le \mathsf{array}[\mathsf{nextToCompare}+1]\}$
35 // from failure of **if** statement on line 20
36 $\boxed{19}\{$ *To be provided* $\}$ // from $\boxed{11}$, $\boxed{18}$
37 }
38 $\boxed{20}\{\forall \mathsf{i} \in [0, \mathsf{nextToCompare}] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{nextToCompare}+1]\}$
39 // from $\boxed{17}$, $\boxed{19}$
40 `nextToCompare = nextToCompare+1;`
41 $\boxed{21}\{\forall \mathsf{i} \in [0, \mathsf{nextToCompare}-1] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{nextToCompare}]\}$
42 // from $\boxed{20}$, assignment on line 40
43 $\boxed{22}\{\forall \mathsf{i},\mathsf{j} \in [\mathsf{lastUnsorted}+1, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}]\}$
44 // from $\boxed{9}$, $\boxed{15}$
45 $\boxed{23}\{\forall \mathsf{i} \in [0, \mathsf{lastUnsorted}] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{lastUnsorted}+1]\}$ // from $\boxed{10}$, $\boxed{14}$
46 }
47 $\boxed{24}\{\mathsf{nextToCompare} = \mathsf{lastUnsorted}\}$ // from failure of **while** test on line 13
48 $\boxed{25}\{\forall \mathsf{i},\mathsf{j} \in [\mathsf{lastUnsorted}+1, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}]\}$
49 // from $\boxed{4}$, $\boxed{22}$
50 $\boxed{26}\{\forall \mathsf{i} \in [0, \mathsf{lastUnsorted}] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{lastUnsorted}+1]\}$ // from $\boxed{5}$, $\boxed{23}$
51 $\boxed{27}\{\forall \mathsf{i} \in [0, \mathsf{nextToCompare}-1] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{nextToCompare}]\}$
52 // from $\boxed{7}$, $\boxed{21}$
53 $\boxed{28}\{\forall \mathsf{i} \in [0, \mathsf{lastUnsorted}-1] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{lastUnsorted}]\}$ // from $\boxed{24}$, $\boxed{27}$
54 $\boxed{29}\{\forall \mathsf{i},\mathsf{j} \in [\mathsf{lastUnsorted}, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}]\}$
55 // from $\boxed{25}$, $\boxed{28}$
56 `lastUnsorted = lastUnsorted-1;`
57 $\boxed{30}\{\forall \mathsf{i},\mathsf{j} \in [\mathsf{lastUnsorted}+1, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}]\}$
58 // from $\boxed{29}$, assignment on line 56
59 $\boxed{31}\{\forall \mathsf{i} \in [0, \mathsf{lastUnsorted}] : \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{lastUnsorted}+1]\}$
60 // from $\boxed{28}$, assignment on line 56
61 }
62 $\boxed{32}\{ \qquad \}$ // from failure of **while** test on line 6
63 $\boxed{33}\{\forall \mathsf{i},\mathsf{j} \in [\mathsf{lastUnsorted}+1, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}]\}$ // from $\boxed{2}$, $\boxed{30}$
64 $\boxed{34}\left\{\begin{array}{l}\forall \mathsf{i} \in [0, \mathsf{array.length}-1] : \mathsf{i} < \mathsf{j} \Rightarrow \mathsf{array}[\mathsf{i}] \le \mathsf{array}[\mathsf{j}] \\ \quad \&\& \ \mathbf{members}(\mathsf{array}) = \mathbf{members}(\mathsf{original})\end{array}\right\}$ // from $\boxed{33}$, $\boxed{32}$, $\boxed{15}$, $\boxed{14}$
65 }
66
67

2.4.2.3 Elucidations The following elucidations should help to make the meanings of the assertions clearer.

$\boxed{1}$ The whole of the array is unsorted — the `lastUnsorted` index is the last index in the array.

$\boxed{2}$ If `i` and `j` are indices in the sorted part of the array, and `i` < `j`, then the array entry at index `i` must be less than or equla to the array entry at index `j`. I.e. the sorted part of the list is sorted

$\boxed{3}$ If `i` is an index in the unsorted part of the array then the array entry at that index cannot be larger than the smallest (i.e. first) entry in the sorted part of the array. I.e. the smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{4}$ The sorted part of the list is sorted

$\boxed{5}$ The smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{6}$ Starting the next bubble — `nextToCompare` is zero

$\boxed{7}$ If `i` is an index in the bubbled part of the array then the array entry at this index must be less than or equal to the final entry in the bubbled part. I.e. the final element in the bubbled part is a maximum value for the bubbled part

$\boxed{8}$ We have not reached the end of the unsorted part yet — i.e. `nextToCompare` — is less than `lastUnsorted`

$\boxed{9}$ The sorted part of the list is sorted

$\boxed{10}$ The smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{11}$ The final element in the bubbled part is a maximum value for the bubbled part

$\boxed{12}$ The next two items — i.e. `array`[`nextToCompare`] and `array`[`nextToCompare`+ `1`] — are out of order

$\boxed{13}$ The two values have been swapped

$\boxed{14}$ The unsorted part of the array still contains the same members as it did before the swap

$\boxed{15}$ *To be provided.*

$\boxed{16}$ The correct ordering has been restored

$\boxed{17}$ The final element in the enlarged bubbled part — now reaching to `nextToCompare`+ `1` — is a maximum value for the bubbled part

$\boxed{18}$ The two elements being compared are in the right order

$\boxed{19}$ The final element in the enlarged bubbled part is a maximum value for the bubbled part

$\boxed{20}$ The final element in the enlarged bubbled part is a maximum value for the bubbled part

$\boxed{21}$ The final element in the bubbled part is a maximum value for the bubbled part

$\boxed{22}$ The sorted part of the list is sorted

$\boxed{23}$ The smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{24}$ Finished this bubble — `nextToCompare` has reached `lastUnsorted`

$\boxed{25}$ The sorted part of the list is sorted

$\boxed{26}$ The smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{27}$ The final element in the bubbled part is a maximum value for the bubbled part

$\boxed{28}$ The final element in the bubbled part of the array, which now covers the whole of the unsorted part, is at least as large as any of the other elements of the bubbled part. I.e. the final entry in the unsorted part is a maximum value for the unsorted part.

$\boxed{29}$ The (expanded) sorted part of the list is sorted

$\boxed{30}$ The sorted part of the list is sorted

$\boxed{31}$ The smallest element in the sorted part is at least as big as any of the elements in the unsorted part.

$\boxed{32}$ We have reached the end of the sort — `lastUnsorted` has passed the start of the array

$\boxed{33}$ The sorted part of the list is sorted

$\boxed{34}$ *To be provided.*

2.4.2.4 Justifications The following are longer justifications for the derivation of the assertions.

$\boxed{1}$ Follows obviously from the assignment on line 2.

$\boxed{2}$ From $\boxed{1}$, `lastUnsorted` = `array.length` − `1`, so the sorted part of the array is empty. This assertion is therefore trivially true, as an empty section of the array is, by definition, sorted.

$\boxed{3}$ From $\boxed{1}$, $\texttt{lastUnsorted} = \texttt{array.length} - 1$, so the sorted part of the array is empty. This assertion is therefore trivially true, as there is no smallest sorted element to compare the unsorted elements with.

$\boxed{4}$ This is true before entering the **while** loop (because of $\boxed{2}$), and at the end of the **while** loop (because of $\boxed{30}$), so it must also be true here.

$\boxed{5}$ *To be provided.*

$\boxed{6}$ Follows obviously from the assignment on line 10.

$\boxed{7}$ *To be provided.*

$\boxed{8}$ The **while** test on line 13 was successful, so this must be true

$\boxed{9}$ This is true before entering the **while** loop (because of $\boxed{4}$), and at the end of the **while** loop (because of $\boxed{22}$), so it must also be true here.

$\boxed{10}$ This is true before entering the **while** loop (because of $\boxed{5}$), and at the end of the **while** loop (because of $\boxed{23}$), so it must also be true here.

$\boxed{11}$ This is true before entering the **while** loop (because of $\boxed{7}$), and at the end of the **while** loop (because of $\boxed{21}$), so it must also be true here.

$\boxed{12}$ This follows from the success of the **if** test on line 20

$\boxed{13}$ This follows from the property 1 of swap.

$\boxed{14}$ *To be provided.*

$\boxed{15}$ From $\boxed{8}$, $\texttt{nextToCompare} < \texttt{lastUnsorted}$, so the swap has not affected the sorted part of the array.

$\boxed{16}$ From $\boxed{12}$ the two values were out of order. From $\boxed{13}$ these values have been swapped, so they must now be in order.

$\boxed{17}$ This requires a bit more detail.

From $\boxed{11}$, before the swap, we have

$$\forall \texttt{i} \in [0, \texttt{nextToCompare} - 1] : \texttt{array}_{\texttt{old}}[\texttt{i}] \leq \texttt{array}_{\texttt{old}}[\texttt{nextToCompare}]. \tag{1}$$

Also, from property 2 of `swap`, the swap does not affect the values up to index $\texttt{nextToCompare} - 1$, so, from this, and from (1) above,

$$\forall \texttt{i} \in [0, \texttt{nextToCompare} - 1] : \texttt{array}_{\texttt{new}}[\texttt{i}] \leq \texttt{array}_{\texttt{old}}[\texttt{nextToCompare}]. \tag{2}$$

From $\boxed{13}$ we have swapped the values at `nextToCompare` and `nextToCompare+1` so, in particular

$$\texttt{array}_{\texttt{new}}[\texttt{nextToCompare} + 1] = \texttt{array}_{\texttt{old}}[\texttt{nextToCompare}]$$

So we can substitute $\texttt{array}_{\texttt{new}}[\texttt{nextToCompare}+1]$ for $\texttt{array}_{\texttt{old}}[\texttt{nextToCompare}]$ in (2) above, to give

$$\forall \texttt{i} \in [0, \texttt{nextToCompare} - 1] : \texttt{array}_{\texttt{new}}[\texttt{i}] \le \texttt{array}_{\texttt{new}}[\texttt{nextToCompare}+1]. \qquad (3)$$

Also $\boxed{16}$ tells us that

$$\texttt{array}_{\texttt{new}}[\texttt{nextToCompare}] < \texttt{array}_{\texttt{new}}[\texttt{nextToCompare} + 1] \qquad (4)$$

I.e., (3) guarantees the inequality up to index $\texttt{nextToCompare} - 1$, and (4) guarantees it for index `nextToCompare`, so

$$\forall \texttt{i} \in [0, \texttt{nextToCompare}] : \texttt{array}_{\texttt{new}}[\texttt{i}] \le \texttt{array}_{\texttt{new}}[\texttt{nextToCompare} + 1].$$

$\boxed{18}$ The **if** test on line 20 failed, so this must be true

$\boxed{19}$ From $\boxed{11}$ $\forall \texttt{i} \in \texttt{array}[0, \texttt{nextToCompare}-1] : \texttt{array}[\texttt{i}] \le \texttt{array}[\texttt{nextToCompare}]$.
From $\boxed{18}$ we have $\texttt{array}[\texttt{nextToCoompare}] \le \texttt{array}[\texttt{nextToCompare} + 1]$
so

$$\begin{aligned} \forall \texttt{i} \in \texttt{array}[0, \texttt{nextToCompare} - 1] : \texttt{array}[\texttt{i}] &\le \texttt{array}[\texttt{nextToCompare}] \\ &\le \texttt{array}[\texttt{nextToCompare} + 1], \end{aligned}$$

which gives the desired result..

$\boxed{20}$ This is true both in the **if** part (because of $\boxed{17}$), and in the **else** part (because of $\boxed{19}$), so it must also be true here.

$\boxed{21}$ Adjusting the values in $\boxed{20}$ to take account of the assignment on line 40 gives the desired result.

$\boxed{22}$ This was true before the swap, from $\boxed{9}$. From $\boxed{15}$ the swap did not change the sorted part of the array, so it must still be true now.

$\boxed{23}$ This was true before the swap, from $\boxed{10}$. From $\boxed{14}$ the swap did not change the members of the unsorted part of the array, so it must still be true now.

$\boxed{24}$ From the failure of the **while** test on line 13 we must have $\texttt{nextToCompare} \ge \texttt{lastUnsorted}$. Since `nextToCompare` only ever increases by one at a time (line 40) we must have $\texttt{nextToCompare} = \texttt{lastUnsorted}$.

$\boxed{25}$ This is true before entering the **while** loop (because of $\boxed{9}$), and at the end of the **while** loop (because of $\boxed{22}$), so it must be true here.

$\boxed{26}$ This is true before entering the **while** loop (because of $\boxed{10}$), and at the end of the **while** loop (because of $\boxed{23}$), so it must be true here.

$\boxed{27}$ This is true before entering the **while** loop (because of $\boxed{7}$), and at the end of the **while** loop (because of $\boxed{21}$), so it must be true here.

$\boxed{28}$ From $\boxed{24}$ we can substitute `lastUnsorted` for `nextToCompare` in $\boxed{27}$ to give the desired result.

$\boxed{29}$ From $\boxed{25}$ we know that the array is sorted from index $\texttt{lastUnsorted} + \texttt{1}$ onwards. From $\boxed{26}$ we also know that $\texttt{array}[\texttt{lastUnsorted}] < \texttt{array}[\texttt{lastUnsorted}+\texttt{1}]$, so the array is now sorted from index `lastUnsorted` onwards.

$\boxed{30}$ Substituting $\texttt{lastUnsorted} + \texttt{1}$ for `lastUnsorted` (because of the assignment on line `56`) in $\boxed{29}$ gives this

$\boxed{31}$ Substituting $\texttt{lastUnsorted} + \texttt{1}$ for `lastUnsorted` (because of the assignment on line `56`) in $\boxed{28}$ gives this

$\boxed{32}$ From the failure of the **while** test on line `6` we must have $\texttt{lastUnsorted} < \texttt{lastUnsorted}$. Since `lastUnsorted` only ever decreases by one at a time (line `56`) we must have $\texttt{lastunsorted} = -\texttt{1}$.

$\boxed{33}$ This was true before entering the **while** loop (because of $\boxed{2}$), and was true at the end of the **while** loop (because of $\boxed{30}$), so it must be true here.

$\boxed{34}$ The sorted part of this property comes from substituting -1 for `lastUnsorted` (because of $\boxed{32}$) in $\boxed{33}$. That the array still contains the same values as it did at the start follows from $\boxed{14}$ and $\boxed{15}$, which guarantee that the swaps do not affect the membership of the array.

3 Improve a correct programme

Derive the `cube` programme from the programme:

```
public static int cube(int n) {
   int cube;
   cube = n³;
   return cube;
}
```

Hint: in the derivation part of this exercise you may find it helpful to define a method `cube` that returns a triple of values:

$$\texttt{cube}(n) = (n^3, 3n^2, 3n)$$

4 Bug Hunt

Despite all the claims in the lecture (and the “proofs”!) the `square` programme is not completely correct! What is wrong with it?

End of correctness tutorial