15-122: Principles of Imperative Computation

Recitation Week 3 — Rob Simmons

In class, we covered one quadratic sort, *selection sort*, and two $O(n \log n)$ sorts, *quicksort* and *mergesort*. To practice with thinking about sorts in terms of loop invariants, today we'll look at two other quadratic sorts, *insertion sort* and *bubble sort*.

Insertion Sort

```
1  void sort(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures is_sorted(A, 0, n);
4  {
5     for (int i = 0; i < n; i++)
6     //@loop_invariant 0 <= i && i <= n;
7     //@loop_invariant is_sorted(A, 0, i);
8     {
9        int j = i;
10
11       while (_____________________________________________________)
12
13       //@loop_invariant _____________________________________________;
14       //@loop_invariant is_sorted(A, 0, j);
15       //@loop_invariant is_sorted(A, j, i+1);
16       //@loop_invariant le_segs(A, 0, j, A, j+1, i+1);
17       {
18          swap(A, ____________________, ____________________);
19          j--;
20       }
21    }
22 }
```

(a) Write correct invariants for line 13 so that the preconditions of the functions on lines 14-16 are satisfied, and prove that these preconditions are satisfied. What other lines do you need?

(b) Assuming the inner loop invariants are correct, prove the outer ones are preserved.

(c) What goes wrong if we're missing just the loop invariant on line 13? 14? 15? 16?

Bubble Sort

Different versions of bubble sort use one or both of two different tricks; one trick avoids avoids re-scanning the upper part of the array, which we know to be sorted, and another trick allows sorting to end early if the array is mostly sorted. The next page has two bubble sorts, each uses one of the tricks.

For the second bubble sort, which is $O(n)$ if given an already-sorted array, our loop invariants allow us to prove that the function is partially correct, but they don't allow us to prove that the function terminates! Make sure you understand why we know that the insertion sort and first bubble sort terminate: there are two loops in each example, and you need to prove that both loops always terminate.

```
1  void sort(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures is_sorted(A, 0, n);
4  {
5     for (int upper = n; upper > 0; upper--)
6
7     //@loop_invariant _____________________________________________;
8     //@loop_invariant le_segs(A, 0, upper, A, upper, n);
9     //@loop_invariant is_sorted(A, upper, n);
10    {
11       for (int j = 0; j < upper-1; j++)
12
13       //@loop_invariant _____________________________________________;
14       //@loop_invariant ge_seg(A[j], A, 0, j);
15       {
16          if (A[j] > A[j+1]) {
17             swap(A, j, j+1);
18          }
19       }
20    }
21 }
```

(a) Write correct invariants for lines 7 and 13 so that the array access on line 14 is safe.

(b) What goes wrong in each example without the invariant on line 8? Line 9? Line 14?

```
1  void sort(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures is_sorted(A, 0, n);
4  {
5     bool done = false;
6     while (!done)
7     //@loop_invariant !done || is_sorted(A, 0, n);
8     {
9        bool swapped = false;
10       for (int j = 0; j < n-1; j++)
11       //@loop_invariant 0 <= j && j <= n;
12       //@loop_invariant swapped || is_sorted(A, 0, j);
13       //@loop_invariant j == n || ge_seg(A[j], A, 0, j);
14       {
15          if (A[j] > A[j+1]) {
16             swapped = true;
17             swap(A, j, j+1);
18          }
19       }
20       if (!swapped) done = true;
21    }
22 }
```

(a) Why do we need to write `j == n` on line 13 in the second example? Why couldn't we write just `ge_seg(A[j], A, 0, j)`? What's wrong with it?

(b) Why can't we change line 11 to be `0 <= j && j < n` and skip the `j == n` on line 13?

(c) Okay, then, why do we need the troublesome line 13 at all?