15-122: Principles of Imperative Computation — Spring 2023

Recitation 04: A Strange Sort of Proof — Thursday February 2nd

Recently, we saw an algorithm that sorts an array in time quadratic in the number n of elements it contains: *selection sort*. In addition, we discussed how to improve this $O(n^2)$ bound to the much faster $O(n \log n)$ bound with *mergesort*. Today, we'll take a look at the correctness proof for a short snippet of this algorithm: the `merge` function. This is a somewhat complex proof, so be sure to follow along carefully!

Merge Code

```
int[] merge(int[] A, int m, int[] B, int n)
//@requires 0 <= m && m <= \length(A);
//@requires 0 <= n && n <= \length(B);
//@requires is_sorted(A, 0, m) && is_sorted(B, 0, n);
//@ensures \length(\result) == m + n;
//@ensures is_sorted(\result, 0, m + n);
{
  int[] C = alloc_array(int, m + n);
  int a_i = 0;  // variable used to index array A
  int b_i = 0;  // variable used to index array B
  int c_i = 0;  // variable used to index array C

  while (a_i < m && b_i < n)
  //@loop_invariant 0 <= a_i && a_i <= m;
  //@loop_invariant 0 <= b_i && b_i <= n;
  //@loop_invariant c_i == a_i + b_i;
  //@loop_invariant 0 <= c_i && c_i <= m + n;
  //@loop_invariant is_sorted(C, 0, c_i);
  //@loop_invariant le_segs(C, 0, c_i, A, a_i, m);
  //@loop_invariant le_segs(C, 0, c_i, B, b_i, n);
  {
    if (A[a_i] < B[b_i]) {
      C[c_i] = A[a_i];
      a_i++;
    } else {
      C[c_i] = B[b_i];
      b_i++;
    }
    c_i++;
  }

  //@assert a_i == m || b_i == n;

  ... // code for filling in the remaining portions of C -- omitted

  return C;
}
```

To proceed, we will largely follow the same four steps we have used all semester to show correctness for a function with a loop. We will make one small modification: for EXIT, to work with our shortened code, we'll be proving the assert on line 32 rather than the postconditions. Below is the structure we will follow.

1. Prove the loop invariants hold INITially
2. Show that the loop invariants are PRESErved
3. Show that the loop TERMinates
4. Prove that the assert on line 32 holds on EXIT

Visualizing the main loop invariants in a diagram will make the rest of the proof much easier to write. Feel free to use this space for this purpose!

Checkpoint 0

Prove that the following loop invariants are initially true.

Line 16: `c_i == a_i + b_i`

A. ______________________________ by ______________________________

B. ______________________________ by ______________________________

C. ______________________________ by ______________________________

D. ______________________________ by ______________________________

Line 18: `is_sorted(C, 0, c_i)`

A. ______________________________ by ______________________________

B. ______________________________ by ______________________________

Checkpoint 1

Next, let's prove the preservation of the loop invariant on line 16.

Assumption: ______________________________

To show: ______________________________

This proof proceeds by cases.

Case `A[a_i] < B[b_i]`:

A. ______________________ by ______________

B. ______________________ by ______________

C. ______________________ by ______________

D. ______________________ by ______________

E. ______________________ by ______________

Case `A[a_i] >= B[b_i]`: *(take home)*

A. ______________________ by ______________

B. ______________________ by ______________

C. ______________________ by ______________

D. ______________________ by ______________

E. ______________________ by ______________

Take-home exercise: Provide an argument that shows that none of the quantities `a_i`, `b_i` and `c_i` can overflow during an arbitrary iteration of the loop.

Checkpoint 2

Now, let's prove the preservation of the loop invariant on line 18. Feel free to use mathematical notation for the **arrayutil** functions, e.g. `x > A[0, lo)` instead of `gt_seg(x, A, 0, lo)`. For clarity, we'll also be using `C′` to refer to the modified array `C` after the loop body is executed.

Assumption: ______________________________

To show: ______________________________

This proof proceeds by cases.

Case `A[a_i] < B[b_i]`:

A. ______________________ by ______________

B. ______________________ by ______________

C. ______________________ by ______________

D. ______________________ by ______________

E. ______________________ by ______________

F. ______________________ by ______________

Case `A[a_i] >= B[b_i]`: *(take home)*

A. ______________________ by ______________

B. ______________________________ by ________________

C. ______________________________ by ________________

D. ______________________________ by ________________

E. ______________________________ by ________________

F. ______________________________ by ________________

Checkpoint 3

Prove that the loop terminates.

On each iteration, one of the integer quantities ________________ or ________________

decreases by ______________________________ and approaches ____________.

The loop will terminate because ______________________________________.

Checkpoint 4

Below, we will complete a modified EXIT proof that proves the `@assert` statement on line 32 as we presented an incomplete version of `merge`.

To show: ____

A. ______________________________ by ________________

B. ______________________________ by ________________

C. ______________________________ by ________________

D. ______________________________ by ________________

We're done! This was just a taste of a correctness proof on one of our more complex algorithms. If you would like more practice, you can download the mergesort code from the course website and try working through the full correctness proof.

In-Place

A function is *in-place* if it allocates a constant amount of memory (possibly none) to carry out its computation. For example, binary search is in-place because it does not allocate any memory. On the other hand, a function that returns a copy of an array passed to it as input is not in-place as it needs to create (and return) an array of the same length as its input — which could be an arbitrary long array.

Is the function `merge` in-place? ◯ Yes ◯ No Why? ______________________________

____

Stable Sorting

A sorting algorithm is *stable* if it preserves the relative order of elements that are the same. Consider the following input array

1_a	5	1_b	2_a	2_b

where we use colors and subscripts to distinguish different occurrences of the same elements. Here are two ways this array could be sorted:

1_a	1_b	2_b	2_a	5

1_a	1_b	2_a	2_b	5

Observe that, in the array on the left, the red 1_a is before the blue 1_b — like in the input array — but unlike in the input array the green 2_b comes before the purple 2_a. A sorting algorithm that produces this array would not be stable. Instead, in the array on the right, the two `1`s and the two `2`s occur in the same order as in the input array. A sorting algorithm that always does this is stable.

For mergesort to be stable, the function `merge` needs to preserve the relative order of the duplicate elements in its input. Specifically, any duplicate element in `A[0,n)` followed by `B[0,m)` should occur in the same order in `C[0,n+m)`.

The given code for `merge` is *not* stable. Give an example that shows this

merge(| | | , | | |) = | | | | |

What simple change needs to be made to this code so that it is stable?