15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 3 Partial Solutions

In this homework assignment, we will work with specifying and implementing search in an array. In lecture, we worked on searching for any integer in an array. In this assignment, we will talk about searching for the *first* occurrence integer in an array with duplicates.

You will use some of the functions from the `arrayutil.c0` library that was discussed in lecture in this assignment.

3. **Linear search**

Now we'll fill in the loop body with code that does linear search.

```
/*   1 */   int search(int x, int[] A, int n)
/*   2 */   //@requires 0 <= n && n <= \length(A);
/*   3 */   //@requires is_sorted(A, 0, n);
/*   4 */   /*@ensures (\result == -1 && !is_in(x, A, 0, n))
/*   5 */            || (0 <= \result && \result < n
/*   6 */                && A[\result] == x
/*   7 */                /* YOUR ANSWER FOR 1(e) */); @*/
/*   8 */   {
/*   9 */     int lower = 0;
/*  10 */     int upper = n;
/*  11 */     while (lower < upper)
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
/*  13 */     //@loop_invariant gt_seg(x, A, 0, lower);
/*  14 */     //@loop_invariant le_seg(x, A, upper, n);
/*  15 */     {
/*  16 */       if (A[lower] == x)
/*  17 */         return lower;
/*  18 */       if (A[lower] > x)
/*  19 */         return -1;
/*  20 */       //@assert A[lower] < x;
/*  21 */       lower = lower + 1;
/*  22 */     }
/*  23 */     //@assert lower == upper;
/*  24 */     return -1;
/*  25 */   }
```

(1) (a) Prove that this loop has to terminate. (What quantity gets smaller every time the loop body runs?)

> **Solution:** The quantity `upper-lower` always decreases. If it is x before the loop body runs then it will be $x - 1$ after the loop body runs.

(3) (b) Prove that, in the case that the code returns on line 17 or 19, the postcondition on lines 4-7 – with your modification from 1(e) – always evaluates to true.

Solution: When we start the loop, we know the following:

- `0 <= n && n <= \length(A)` by the function's precondition (line 2, `A` and `n` are never modified by the function)
- `A[0,n)` SORTED by the function's precondition (line 3, `A` and `n` are never modified by the function and `A` is not written to anywhere)
- `lower < upper` by the loop guard (line 11)
- `0 <= lower && lower <= upper && upper <= n` by the first loop invariant (line 12)
- `x > A[0, lower)` by the second loop invariant (line 13)
- `x <= A[upper, n)` by the third loop invariant (line 14)

If we return on line 17, we also know `A[lower]` is `x` by line 16. We therefore want to prove the second part of the postcondition:

- `0 <= \result` by line 17 (`\result` = `lower`) and line 12 (`0 <= lower`)
- `\result < n` by line 17 (`\result` = `lower`), line 11 (`lower < upper`), and line 12 (`upper <= n`)
- `A[\result]` = `x` by line 17 (`\result` = `lower`) and line 16 (`A[lower]` = `x`)
- `!is_in(x, A, 0, \result)` by line 17 (`\result` = `lower`) and line 13 (`x > A[0, lower)` implies `x` $\notin$ `A[0, lower)`)

If we return on line 19, we also know `A[lower] > x` by line 18. We therefore want to prove the first part of the postcondition:

- `\result` = `-1` by line 19
- `x` $\notin$ `A[0,lower)` by line 13 (`x > A[0, lower)` implies `x` $\notin$ `A[0, lower)`)
- `x` $<$ `A[lower]` by line 18
- `x` $<$ `A[lower,n)` by the above and line 3 (`A[0,n) sorted`).
- `x` $\notin$ `A[lower,n)` by the above
- `x` $\notin$ `A[0,n)` by pasting together the two segments that `x` is not in.

4. **Code revisions**

(1) (a) Complete this simpler loop invariant for the code on page 6 by writing a line that tells you something about `upper`. The resulting loop invariant should be true initially, should be preserved by any iteration of the loop, and should allow you to prove the postcondition *without* the modifications you made in 2(c). (You don't have to write the proof.)

Solution:

```
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper;

/*  13 */     //@loop_invariant gt_seg(x, A, 0, lower);

/*  14 */     //@loop_invariant upper == n;
```

(1) (b) Here's an alternate loop body that does perform binary search. You can use it as a replacement for lines 15-22 on page 6:

```
/*  15 */     {
/*  16 */       int mid = lower + (upper-lower)/2;
/*  17 */       if (A[lower] == x) return lower;
/*  18 */       if (A[mid] < x) lower = mid+1;
/*  19 */       else { /*@assert(A[mid] >= x); @*/
/*  20 */         upper = mid;
/*  21 */       }
/*  22 */     }
```

Show that your answer for 4(a) above is *not* a loop invariant of a loop with *this* body. Give specific values for all variables such that `n` and `A` satisfy the preconditions, the loop guard `lower < upper` evaluates to true, and your loop invariants from 4(a) evaluate to true before this loop body runs, but those loop invariants do not evaluate to true after this loop body runs.

Solution:

- `x = 42`
- `A = {0, 1, 42, 42, 90}`
- `n = 5`
- `lower = 0`
- `upper = 5`

In the loop body, `mid` will be set to `2`. We will fail the conditionals on lines `17` and `18` because `A[2] >= 42`, thus after the loop runs `upper` will be `2`, failing the loop invariant we wrote above.