15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 3

Due: Thursday, September 19, 2013 at 10pm

Name: ______________________________

Andrew ID: ______________________________

Recitation: ______________________________

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.

Question	Points	Score
1	5	
2	4	
3	4	
4	2	
Total:	15	

You must do this assignment in one of two ways:

1) Write your answers *neatly* on this PDF, and then submit the stapled printout to the handin box Thursday before lecture or on Thursday afternoon outside of Tom Cortina's office (GHC 4117).

2) Use the TeX template at `https://whiteboard.ddt.cs.cmu.edu/#/15122-f13/assessments/78/1` and either submit your solutions electronically or print them out and hand them in.

1. **Preconditions and postconditions**

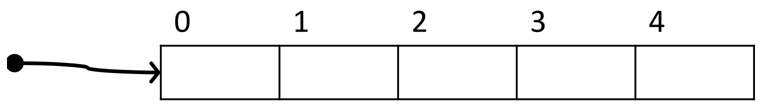
Here is our initial, buggy specification of search for the first occurrence of `x` in an array.

```
/*   1 */   int search(int x, int[] A, int n)
/*   2 */   //@requires 0 <= n && n <= \length(A);
/*   4 */   /*@ensures (\result == -1 && !is_in(x, A, 0, n))
/*   5 */           || (0 <= \result && \result < n
/*   6 */               && A[\result] == x
/*   7 */               && A[\result-1] < x); @*/
```

(1) (a) Give *specific* values of inputs and output such that the precondition evaluates to true and checking the postcondition will cause an array-out-of-bounds exception.

Solution:

- `x = 42`
- `A =`



- `n = 5`
- `\result =`

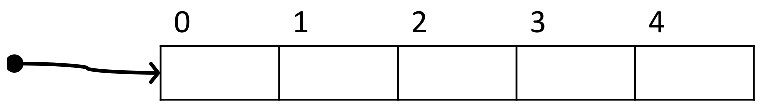
(1) (b) Even in cases where we don't access an array out of bounds, the specification is buggy because we left off an important precondition that should have appeared as line 3: `@requires is_sorted(A, 0, n)`.

Of course, we can perform linear search on an unsorted array, but if our array is not necessarily sorted, line 7 doesn't actually enforce that `\result` is the first occurrence of `x` in the array.

Give specific values for inputs and output such that the precondition on line 2 and the postcondition both evaluate to true but the result is not the index of the first occurrence of `x` in the array. (Hint: the array `A` should not be sorted.)

Solution:

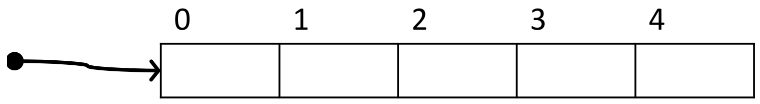
- `x = 42`
- `A =`



- `n = 5`
- `\result =`

(1) (c) Give specific values for inputs and output such that the precondition on line 2 evaluates to true, the postcondition evaluates to *false*, and the result *is* the index of the first occurrence of `x` in the array. (Hint: the array `A` should again not be sorted.)

Solution:

- `x = 42`
- `A =` 

- `n = 5`
- `\result =`

(1) (d) Edit line 7 slightly so that, *if* we require that the array is sorted, the postcondition for `search` is safe and correct. Do *not* use any of the `arrayutil.c0` specification functions.

Solution:

```
/*  7 */
```

(1) (e) Edit line 7 so that *whether or not* we require that the array is sorted, the postcondition for `search` is safe and correct. You'll need to use one of the `arrayutil.c0` specification functions.

Solution:

```
/*  7 */
```

2. **The loop invariant**

   Now that our specification is correct, we'll move on to an also-buggy implementation.

```
/*   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 */     {

...

/* m-3 */     }
/* m-2 */     //@assert lower == upper;
/* m-1 */     return -1;
/*   m */   }
```

   You should assume that the missing loop body does not write to the array `A` or modify the assignable variables `x`, `A`, or `n`.

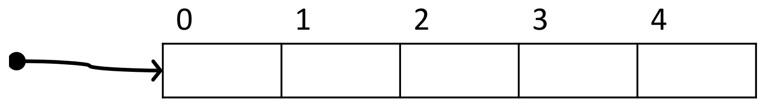
(1) (a) Prove that the loop invariant (lines 12-14) holds initially:

Solution:

(1) (b) This loop invariant does not imply the postcondition when the function exits on line `m-1`! Give specific values for all variables such the precondition evaluates to true, the loop guard evaluates to false, the loop invariants evaluate to true, and the postcondition evaluates to false, given that `\result = -1`.

Solution:

- `x = 42`
- `A =`



- `n = 5`
- `lower =`
- `upper =`

(2) (c) Modify the code *after* the loop so that, if the loop terminates, the postcondition will always be true.

Take care to ensure that any array access you make is safe! You know that the loop invariants on lines `12-14` are true, and you know that the loop guard is false (which, together with the first loop invariant on line `12`, justifies the assertion `lower == upper`).

Solution:

```
  /* Loop ends here... */
  //@assert lower == upper;



  if (________________________________________________) {



      return ___________________________________________;
  }
  return -1;
}
```

3. **Linear search**

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

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
        || (0 <= \result && \result < n
            && A[\result] == x
            /* YOUR ANSWER FOR 1(e) */); @*/
{
  int lower = 0;
  int upper = n;
  while (lower < upper)
  //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
  //@loop_invariant gt_seg(x, A, 0, lower);
  //@loop_invariant le_seg(x, A, upper, n);
  {
    if (A[lower] == x)
      return lower;
    if (A[lower] > x)
      return -1;
    //@assert A[lower] < x;
    lower = lower + 1;
  }
  //@assert lower == upper;
  return -1;
}
```

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

Solution:

(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:

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:

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:

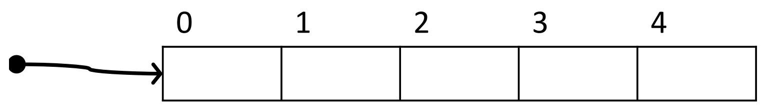
```
/*  12 */     //@loop_invariant 0 <= lower && lower <= upper;

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

/*  14 */     //@loop_invariant
```

(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 =` 

- `n = 5`
- `lower =`
- `upper =`