15-122 : Principles of Imperative Computation, Spring 2016

Written Homework 3

Due: Monday 1st February, 2016

Name: ____

Andrew ID: ______________________________________

Section: _______________________________________

This written homework covers specifying and implementing search in an array and how to reason with contracts. You will use some of the functions from the arrayutil.c0 library that was discussed in lecture in this assignment.

Print out this PDF double-sided, staple pages in order,
and write your answers *neatly* by hand in the spaces provided.

The assignment is due by 1:30pm on Monday 1st February, 2016.

You can hand in the assignment at the beginning of class.

You must hand in your homework yourself;
do not give it to someone else to hand in.

1. Debugging preconditions and postconditions

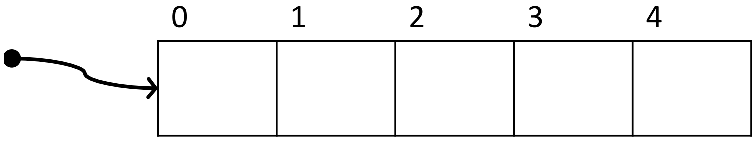
Here is an initial, buggy specification of `search` for the first occurrence of `x` in an array. You should assume the `search` function does not modify the contents of the array `A` in any way.

```
int search(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
// (nothing to see here)
/*@ensures (\result == -1 && !is_in(x, A, 0, n))
      || (0 <= \result && \result < n
         && A[\result] == x
         && A[\result-1] < x); @*/
```

1pt

(a) Give values of `A` and `\result` below, such that the precondition evaluates to true and checking the postcondition will cause an array-out-of-bounds exception.

- `x = 122`
- `A =`



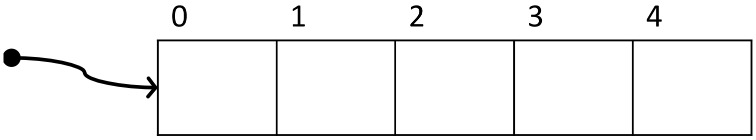
- `n = 5`
- `\result =`

1pt

(b) Notice that the postcondition seems to be relying on `A` being sorted, although the precondition does not specify this. It might be possible, then, that unsorted input will reveal additional bugs in our initial specification.

Give values for `A` and `\result` below, such that `\result != -1`, the precondition and the postcondition both evaluate to true, and `\result` is *not* the index of the first occurrence of `x` in the array.

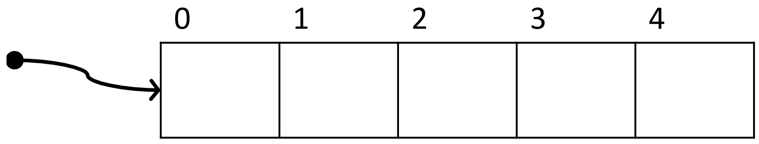
- `x = 122`
- `A =`



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

1pt (c) Give values for A and \result below, such that the precondition evaluates to true, the postcondition evaluates to *false*, and \result *is* the index of the first occurrence of x in the array.

- x = 122
- A =



- n = 5
- \result =

1pt (d) Edit line 7 slightly so that, if we added an additional precondition

```
3 //@requires is_sorted(A, 0, n);
```

the postcondition for search would be safe and it would correctly enforce that A[\result] is the first occurrence of x in A. Do *not* use any of the arrayutil.c0 specification functions.

The addition you make to the postcondition should run in constant time ($O(1)$). (We don't usually care about the complexity of our contracts, of course, but this limits what kinds of answers you can give. In the future, unless we specifically say otherwise, you can assume that the efficiency of contracts doesn't matter.)

```
7 ____________________________________________________
```

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

```
7 ____________________________________________________
```

2. The Loop Invariant

Now we will consider a buggy implementation with a correct specification.

```
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(d) */); @*/
{
  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);
  {
    ...
  }
  //@assert lower == upper;
  return -1;
}
```

You should assume that the missing loop body does not write to the array `A` or modify the locals `x`, `A`, or `n`, but that it might modify `lower` or `upper`.

1pt

(a) In one sentence, explain why `gt_seg(x, A, 0, 0)` and `le_seg(x, A, n, n)` are always `true`, assuming `0 <= n && n <= \length(A)`. Your answer should involve the size of the array segment being tested.

1pt

(b) Prove that the loop invariants (lines 12–14) hold initially.

`0 <= lower` is true because of line(s): ______________

`lower <= upper` is true because of line(s): ______________

`upper <= n` is true because of line(s): ______________

`gt_seg(x, A, 0, lower)` is true because of line(s): ______________

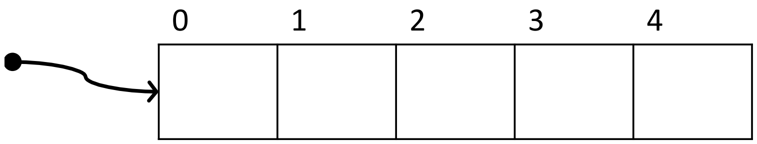
`le_seg(x, A, upper, n)` is true because of line(s): ______________

Take for granted that all the loop invariants are known to be safe. You do need line `n <= \length(A)` from line 2 to reason that the last loop invariant involving `le_seg` is safe (that it satisfies its preconditions). You don't need to include line 2 in your proof that `le_seg(x,A,upper,n)` always evaluates to true.

1pt

(c) Danger! These loop invariants do not imply the postcondition when the function exits on line 24. Give specific values for `A`, `lower`, and `upper` 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`.

- `x = 122`
- `A =`



- `n = 5`
- `\result =`
- `lower =`
- `upper =`

2pts

(d) Modify the code *after* the loop so that, if the loop terminates, the postcondition will always be true. The conditional and the return statement should both run in constant time ($O(1)$) and should not use `arrayutil.c0` specification functions.

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`).

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

25   if (_______________________________________________)

27     return __________________________________________;

29   return -1; // old line 24
30 }            // old line 25
```

3. Code Revisions

Here is a loop body that performs linear search. You can use it as an implementation for lines 15–22 on page 3:

```
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   }
```

1pt

(a) For the loop invariants to hold for this loop body, they must be preserved through each iteration.

Prove that the invariant in line 12 on page 3 is preserved by this loop body.

1pt (b) Prove that the invariant in line 13 is preserved by this loop body. (The proof for line 14 is not required for this answer.)

1pt (c) You might have noticed in the previous part that `upper` does not actually change during the loop, even though all our reasoning assumes it might. So now, complete this simpler loop invariant for the modified code by writing a line that tells you something about `upper`. The resulting loop invariant should be simple, 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(d). (You don't have to write the proof.)

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

13  //@loop_invariant lower == 0 || x > A[lower - 1];

14  //@loop_invariant _________________________________;
```

2pts 4. **Timing code**

In this class we're mostly interested in the big-O behavior of a function, but in the right circumstances it would be possible to come up with a specific function describing the amount of time (say, in milliseconds) that it takes to run a given function.

For this question, consider a C0 function `mystery` with three integer arguments x, y, and z and with a running time in milliseconds that is precisely specified by $T(x, y, z) = c \times x^2 \times y \times 2^z$ for some positive constant c. (We don't know or care what `mystery` is actually computing for the purpose of this question.)

Say, for some values of x and y and z, the function `mystery` takes 1 second to run.

Leaving y and z the same, how would we change the first input (in terms of x) to make the function run for 16 seconds?

$T(\underline{\hspace{6em}}, y, z) = 16$ seconds

Leaving x and z the same, how would we change the second input (in terms of y) to make the function run for 16 seconds?

$T(x, \underline{\hspace{6em}}, z) = 16$ seconds

Leaving x and y the same, how would we change the third input (in terms of z) to make the function run for 16 seconds?

$T(x, y, \underline{\hspace{6em}}) = 16$ seconds

These kinds of calculations allow us to experimentally investigate the Big-O complexity of a function by modifying the size of the input (say, doubling it) and seeing how the running time changes. But one must be careful! If there was an extra two-second constant cost, so that the running time of the `mystery` function was actually $T(x, y, z) = c \times x^2 \times y \times 2^z + 2000$, then `mystery` would have *the same big-O behavior*, but investigating the big-O running time by doubling the first argument from x to $2x$ in the example above could lead us astray!

Why? What additional test(s) might save us from this mistake?