

# 15-122: Principles of Imperative Computation, Spring 2024

## Written Homework 3

Due on **Gradescope**: Monday 29<sup>th</sup> January, 2024 by 9pm

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 discussed in lecture in this assignment.

**Preparing your Submission** You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- [Kami](#), [Adobe Acrobat Online](#), or [DocHub](#), some web-based PDF editors that work from anywhere.
- [Acrobat Pro](#), installed on all [non-CS cluster machines](#), works on many platforms.
- [iAnnotate](#) works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

**Please do not add, remove or reorder pages.**

**Caution** Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

**Submitting your Work** Once you are done, submit this assignment on [Gradescope](#). *Always check it was correctly uploaded.* You have unlimited submissions.

Question:	1	2	3	4	Total
Points:	5	3.5	4.5	2	15
Score:					

### 1. Debugging Preconditions and Postconditions

Here is an initial, buggy specification of the function `find` that returns the index of the first occurrence of an element `x` in an array `A`. You should assume the `find` function does not modify the contents of the array `A` in any way.

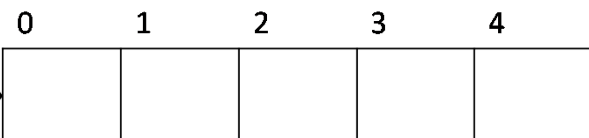
```

1 int find(int x, int[] A, int n)
2 //@requires 0 <= n && n <= \length(A);
3 // (nothing to see here)
4 /*@ensures (\result == -1 && !is_in(x, A, 0, n))
5           || (0 <= \result && \result < n
6             && A[\result] == x
7             && A[\result-1] < x); @*/

```

1pt

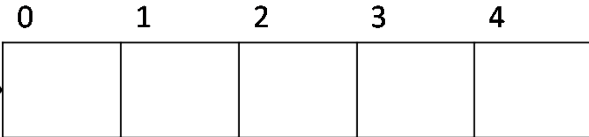
1.1 Give values of `A` and `\result` below, such that the precondition evaluates to `true` and checking the postcondition will cause an array-out-of-bounds error.

- `x = 743`
- `A =` 
- `n = 5`
- `\result = _____`

1pt

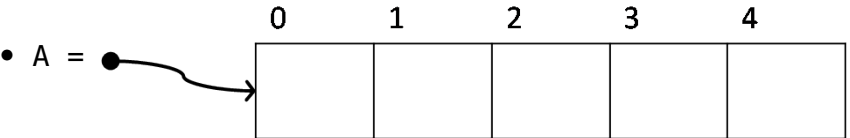
1.2 Notice that the postcondition seems to be relying on `A` being sorted, although the precondition does not specify this. It might be possible, then, that an unsorted input will reveal additional bugs in our initial specification.

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 = 743`
- `A =` 
- `n = 5`
- `\result = _____`

1pt

- 1.3 Give values for  $A$  and  $\backslash result$  below, such that the precondition evaluates to true, the postcondition evaluates to *false*, and  $\backslash result$  is the index of the first occurrence of  $x$  in the array.

- $x = 743$
- $A =$  
- $n = 5$
- $\backslash result =$  \_\_\_\_\_

1pt

- 1.4 Edit line 7 so that the postcondition for `find` is safe and correct *even if the array is not sorted*. Make the answer as simple as possible. You'll need to use one of the `arrayutil.c0` specification functions found at <https://cs.cmu.edu/~15122/handouts/code/arrayutil.c0>.

```
7 _____; @*/
```

If we did have a sorted array, the original line 7 would be *almost* correct.

1pt

- 1.5 Edit the original line 7 slightly so that, if we added an additional precondition

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

the postcondition for `find` would be safe and it would correctly enforce that  $A[\backslash result]$  is the first occurrence of  $x$  in  $A$ . This time, 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 && ( _____ );
```

## 2. The Loop Invariant

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

```
1 int find(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 TO TASK 1.5 */); @*/
8 {
9   int lo = 0;
10  int hi = n;
11  while (lo < hi)
12    //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
13    //@loop_invariant gt_seg(x, A, 0, lo);
14    //@loop_invariant le_seg(x, A, hi, n);
15    {
16      ...
22  }
23  //@assert lo == hi;
24  return -1;
25 }
```

You should assume that the missing loop body does not write to the array  $A$  or modify the local variables  $x$ ,  $A$ , or  $n$ , but that it might modify  $lo$  or  $hi$ .

0.5pts

2.1 In one precise sentence, explain why  $gt\_seg(x, A, 0, 0)$  and  $le\_seg(x, A, n, n)$  are always true, assuming  $0 \leq n \leq \text{length}(A)$ .

1pt

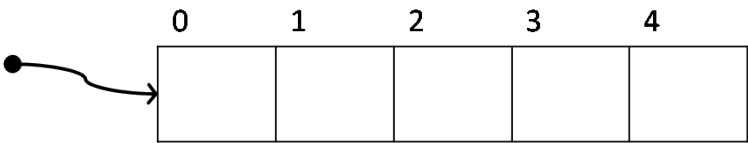
2.2 Prove that the loop invariants (lines 12–14) hold initially.

You may take for granted that all the loop invariants are known to be safe.

$0 \leq lo$	is true because of line(s) _____
$lo \leq hi$	is true because of line(s) _____
$hi \leq n$	is true because of line(s) _____
$gt\_seg(x, A, 0, lo)$	is true because of line(s) _____
$le\_seg(x, A, hi, n)$	is true because of line(s) _____

1pt

2.3 Danger! These loop invariants do not imply the postcondition when the function exits on line 24. Give specific values for  $A$ ,  $lo$ , and  $hi$  such that 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  $\backslash result == -1$ .

- $x = 743$
- $A =$  
- $n = 5$
- $\backslash result = -1$
- $lo =$  \_\_\_\_\_
- $hi =$  \_\_\_\_\_

1pt

- 2.4 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 `lo == hi`).

```
22  /* Loop ends here... */
23  //@assert lo == hi;
24
25  if ( _____ )
26
27      return _____ ;
28
29  return -1;    // old line 24
30 }
```

## 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[lo] == x) return lo;
17   if (A[lo] > x) return -1;
18   //@assert _____;
19   lo = lo + 1;
20 }
21 //@assert lo == hi;

```

1.5pts

3.1 For the loop invariants to hold for this loop body, they must be preserved through each iteration. Prove that the invariant on line 12 on page 3 is preserved by this loop body — you may not need all the provided lines.

A	_____	assumption
B	_____	by _____
C	_____	by _____
D	_____	by _____
E	_____	by _____
F	_____	by _____
G	_____	by _____
Therefore we conclude that		
	_____	by _____

0.5pts

3.2 Fill in the assertion on line 18 with the *strictest fact* about the relationship between  $A[lo]$  and  $x$  that is necessarily true at this point of the execution. Prove that it is true by point-to reasoning.

18	//@assert _____ ; // by _____
----	-------------------------------

1.5pts

3.3 Prove that the loop invariant in line 13 is preserved by this loop body. Again, you may take for granted that it is safe. You may use your answer to the previous task if you wish. (*You may not need all the provided lines.*)

A	_____ assumption
B	_____ by _____
C	_____ by _____
D	_____ by _____
E	_____ by _____
F	_____ by _____
	Therefore we conclude that
	_____ by _____



1pt

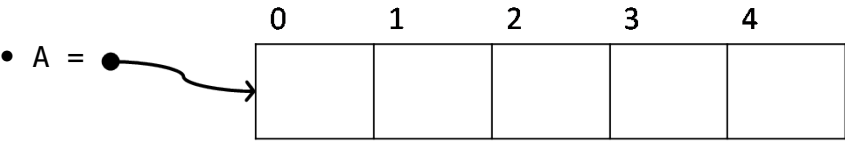
3.4 You might have noticed in the previous part that  $hi$  does not actually change during the loop, even though all our reasoning assumes it might. Could we replace the loop invariant on line 14 with  $hi == n$ ?

To show that this isn't always sufficient, consider an alternate loop body that performs binary search. It replaces the code at the beginning of this question.

```

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

Show that  $hi == n$  is *not* a valid 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  $lo < hi$  evaluates to true, and your loop invariants from the previous question evaluate to true before this loop body runs, but this new loop invariant evaluates to false after one iteration of the loop. Then write the values of  $lo'$  and  $hi'$  after one iteration of the loop.

- $x = 743$
- $A =$  
- $n = 5$
- $lo =$  \_\_\_\_\_
- $hi =$  \_\_\_\_\_
- $lo' =$  \_\_\_\_\_
- $hi' =$  \_\_\_\_\_

#### 4. Code Performance

In this class we're mostly interested in how long code takes to run for asymptotically large inputs, but in the right circumstances it is possible to come up with a specific cost function describing the amount of time that code takes to run.

1pt

4.1 In this task, consider a  $O$  function mystery with three integer arguments  $x$ ,  $y$ , and  $z$ . Its running time in seconds is known to be precisely specified by the cost function  $T(x, y, z) = c \times x^2 \times y \times 2^z$  for some positive constant  $c$ . (We don't know or care what mystery is actually computing.)

For some fixed values of  $x$  and  $y$  and  $z$ , the function mystery takes exactly 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?

$$T(\text{_____}, y, z) = 16 \text{ seconds}$$

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?

$$T(x, \text{_____}, z) = 16 \text{ seconds}$$

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?

$$T(x, y, \text{_____}) = 16 \text{ seconds}$$

1pt 4.2 Consider the following three functions:

```
void f1(int[] A, int n)
//@requires \length(A) == n;
{
    for (int i = 0; i < n; i++)
        A[i] = 15;
}
```

```
int[] f2(int n) {
    int len = 122;

    int[] B = alloc_array(int, len);

    for (int i = 0; i < len; i++) {
        B[i] = n;
    }

    return B;
}
```

```
void f3(int[] C, int n)
//@requires \length(C) == n;
{
    for (int i = 0; i < n; i++) {
        int value = i + 5;
        value += 150;
        value -= 210;
        value = value >> 4;
        value = value % 122;
        C[i] = value / 7;
    }
}
```

For each of these three functions, you write a `main` that calls it with the same very large value of the parameter `n`. You compile each of the three resulting programs without `-d` and run it. Which of them will take the shortest time to run? Concisely explain why.

Which function will run fastest for a very large `n`?

f1

f2

f3

because \_\_\_\_\_  
\_\_\_\_\_