15-122: Principles of Imperative Computation, Fall 2022

Written Homework 3

Due on Gradescope: Monday 19th September, 2022 by 9pm EDT

Name: Arda Akinci

Andrew ID: aakinci

Section: I

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:

- ***PDFescape*** or ***DocHub***, two 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.*

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.

```
int find(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

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 exception.

-x can't be in A

- `x = 729`
- `A =`

0	1	2	3	4
729	800	900	1000	1100

- `n = 5`
- `\result =` 0

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 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.

Ø

- `x = 729`
- `A =`

0	1	2	3	4
780	729	4	729	16

- `n = 5`
- `\result =` 3

1pt

1.3 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 = 729`
- `A =` ●→

0	1	2	3	4
1	2	1000	784	4

- `n = 5`
- `\result =` 3

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/code/arrayutil.c0`.

```
7   !is_in(x, A, \result)  ; @*/
```

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[\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   && ( \result ==0 | x> A[\result - 1] );
```

© Carnegie Mellon University 2022

2. The Loop Invariant

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

```
int find(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 TO TASK 1.5 */); @*/
{
  int lo = 0;
  int hi = n;
  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= n;
  //@loop_invariant gt_seg(x, A, 0, lo);
  //@loop_invariant le_seg(x, A, hi, n);
  {
    ...
  }
  //@assert lo == hi;
  return -1;
}
```

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 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.

The size of both arrays being tested is equal to 0, which means it must be that x > A[0,0) and x ≤ A[n,n), respectively

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. 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,hi,n)` always evaluates to `true`.

`0 <= lo` is true because of line(s) 9

`lo <= hi` is true because of line(s) 2,9,10

`hi <= n` is true because of line(s) 10

`gt_seg(x, A, 0, lo)` is true because of line(s) 9

`le_seg(x, A, hi, n)` is true because of line(s) 10

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 `\result == -1`.

- lo ≥ hi
- sorted
- x in A

- `x = 729`
- `A =`

0	1	2	3	4
5	20	729	780	900

- `n = 5`
- `\result = -1`
- `lo =` 2
- `hi =` 2

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;

25   if (  A[lo] == x  )

27       return  lo  ;

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

© Carnegie Mellon University 2022

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 0 <= lo && lo <= hi && hi <= n assumption

B lo < hi by line 11

C lo' = lo + 1 by line 19

D lo + 1 ≤ hi by math, line 11

E lo' ≤ hi by assumption, line 19

F 0 ≤ lo' by assumption, line C

G ______ by ______

Therefore we conclude that

0 ≤ lo' && lo' ≤ hi' && hi <= n by proof above

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 A[lo] < x ; // by 16, 17
```

1.5pts

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

WTS x > A[0, lo')

A gt-seg(x, A, 0, lo) assumption

B x > A[0, lo) by assumption

C x > A[lo] by 16, 17 (negation)

D x > A[0, lo+1) by C, B

E gt-seg(x, A, 0, lo+1) by D

F by

Therefore we conclude that

gt-seg(x, A, 0, lo') by preservation

```
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  {
```

© Carnegie Mellon University 2022

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.

```
{
  int mid = lo + (hi-lo)/2;
  if (A[lo] == x) return lo;
  if (A[mid] < x) lo = mid+1;
  else { //@assert A[mid] >= x;
    hi = mid;
  }
}
```

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 = 729`
- `A =`

0	1	2	3	4
726	729	730	731	732

- `n = 5`
- `lo =` 0 — `lo' =` 0
- `hi =` 5 — `hi' =` 2

lo < hi:

© Carnegie Mellon University 2022

4. **Timing Code**

The following run times were obtained when using two different algorithms on a data set of size n. You are asked to extrapolate the asymptotic complexity of these algorithms based on this time data. Determine the asymptotic complexity of each algorithm as a function of n. Use big-O notation in its tightest form and briefly explain how you reached the conclusion.

1pt **4.1**

n	Execution Time
1000	0.564 milliseconds
2000	2.271 milliseconds
4000	8.992 milliseconds
8000	36.150 milliseconds

Asymptotic complexity: $O(\underline{n^2})$

When you double the size of n, the execution time is quadrupled. For exmple, if n=1000 then if you want to check for n=2000 then you do $O((2n)^2) = O(4n)$ which is four times the exectime time of original n.

1pt **4.2**

n	Execution Time
1000	0.043 milliseconds
1000000	43.68 milliseconds
1000000000	43960.55 milliseconds

Asymptotic complexity: $O(\underline{n})$

$$\frac{43.68}{0.043} \approx 1000 \qquad \frac{1000000}{1000} = 1000$$

When n multiplied by 1000 in size, its execution time also is multiplied by 1000 => linear complexity

© Carnegie Mellon University 2022