15-122: Principles of Imperative Computation, Spring 2022

Written Homework 3

Due on Gradescope: Monday 7th February, 2022 by 9pm EST

Name: Greg Budhijanto

Andrew ID: gbudhija

Section: R

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**

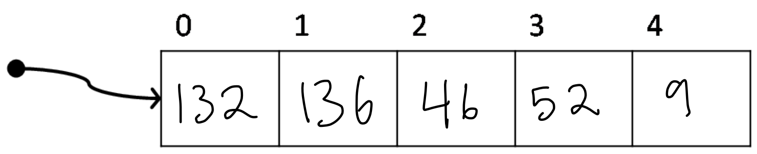
   Here is an initial, buggy specification of the function `search` that returns the index of the first occurrence of an element `x` in an array `A`. You should assume the `search` function does not modify the contents of the array `A` in any way.

```
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

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 = 131`
- `A =`



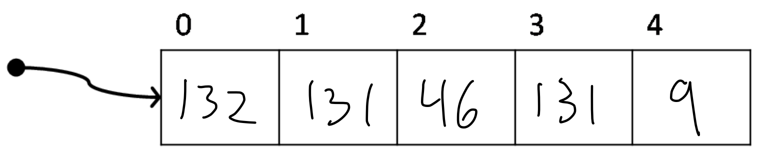
- `n = 5`
- `\result =` -1

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.

- `x = 131`
- `A =`

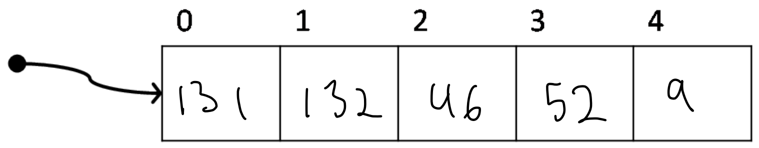


- `n = 5`
- `\result =` 3

1pt

1.3 Give values for `A` and `\result` below, such that the precondition evaluates to `true`, the postcondition evaluates to *`false`*, and <u>`\result` *is* the index of the first occurrence of x in the array</u>.

- `x = 131`
- `A =`



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

1pt

1.4 Edit line 7 so that the postcondition for `search` is safe and correct <u>*even if the array not is sorted*</u>. 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_sorted(A,0,n) || \result == 0 ; @*/
```

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

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 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.

Both functions gt_seg(x,A,0,0) and le_seg(x,A,n,n) meet the precondition (same for both) that the high value <= \length of the array and the first statement of the body (also same for both) that if the low value == the high value, return true.

1pt

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

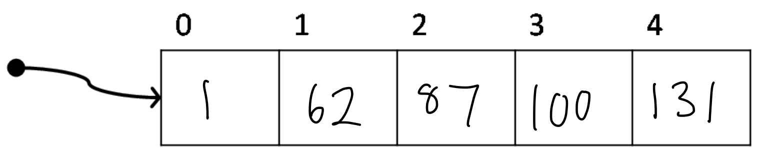
You may take for granted that all the loop invariants are known to be safe. You do need line `n <= \length(A)` from line 2 to reason that the last loop invariant involving `le_seg` is safe (that it satisfies its preconditions). You don't need to include line 2 in your proof that `le_seg(x,A,hi,n)` always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

`0 <= lo`	is true because of line(s)	9
`lo <= hi`	is true because of line(s)	9, 10, 2
`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`, <u>the loop guard evaluates to `false`</u>, the loop invariants evaluate to `true`, and the postcondition evaluates to `false`, given that `\result == -1`.

- `x = 131`
- `A =`



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

```
bool le_seg(int x, int[] A, int lo, int hi)
/*@requires 0 <= lo
          && lo <= hi
          && hi <= \length(A); @*/
{
  if (lo == hi) return true;
  return x <= A[lo] && le_seg(x, A, lo+1, hi);
}
```

assumes sorted

```
bool gt_seg(int x, int[] A, int lo, int hi)
/*@requires 0 <= lo
          && lo <= hi
          && hi <= \length(A); @*/
{
  if (lo == hi) return true;
  return x > A[lo] && gt_seg(x, A, lo+1, 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;

25   if ( A[hi]==x )

27       return hi ;

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

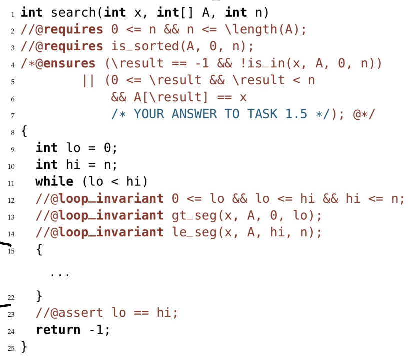
```
 1 int search(int x, int[] A, int n)
 2 //@requires 0 <= n && n <= \length(A);
 3 //@requires is_sorted(A, 0, n);
 4 /*@ensures (\result == -1 && !is_in(x, A, 0, n))
 5          || (0 <= \result && \result < n
 6              && A[\result] == x
 7              /* YOUR ANSWER TO TASK 1.5 */); @*/
 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   {
       ...
22   }
23   //@assert lo == hi;
24   return -1;
25 }
```

lo = 0   hi = 5   0 < 5   1, 62, 87, 100, 131   x = 131

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  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 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   {

       ...

22   }
23   //@assert lo == hi;
24   return -1;
25 }
```

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 $0 \le lo$ && $lo \le hi$ && $hi \le n$ assumption

B $hi' = hi$ and $n = n$ by remains unchanged

C $hi = n$ therefore $hi' = n'$ by line 10 and B

D $lo' = lo + 1$ and $lo = 0$, therefore $lo' \ge 1 > 0$ by line 19, 9, and math

E $lo < hi$ and $lo = lo' - 1$, therefore $lo' - 1 < hi$ by line 11, 19, and math

F $hi' = hi$, therefore $lo' - 1 < hi'$ by B, and math

G $lo' = lo + 1$, therefore $lo' \le hi$ by line 19, E, F, and assumption

Therefore we conclude that

$0 \le lo'$ && $lo' \le hi'$ && $hi' \le n'$ by D, E, F, B, and C

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 negation of lines 16 & 17
```

```
bool le_seg(int x, int[] A, int lo, int hi)
/*@requires 0 <= lo
         && lo <= hi
         && hi <= \length(A); @*/
{
  if (lo == hi) return true;
  return x <= A[lo] && le_seg(x, A, lo+1, hi);
}
```

assumes sorted

```
bool gt_seg(int x, int[] A, int lo, int hi)
/*@requires 0 <= lo
         && lo <= hi
         && hi <= \length(A); @*/
{
  if (lo == hi) return true;
  return x > A[lo] && gt_seg(x, A, lo+1, hi);
}
```

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.)*

A gt_seg(x, A, 0, lo) = true assumption

B is_sorted(A, 0, n) = true by line 2, remains unchanged

C lo' = int hi' in gt_seg function by line 13

D int lo' = 0 and increments by 1 recursively by line 13 & gt_seg line 7

E recursive call returns true when int lo' == int hi' == lo' by gt_seg line 6

F by

Therefore we conclude that

gt_seg (x, A, 0, lo') = true by

0	1	2	3	4
24	39	42	106	131

```
                                    0        lo
1 bool gt_seg(int x, int[] A, ~~int lo~~, ~~int hi~~)
2 /*@requires 0 <= lo
3           && lo <= hi
4           && hi <= \length(A); @*/
5 {
6   if (lo == hi) return true;
7   return x > A[lo] && gt_seg(x, A, lo+1, hi);
8 }
```

```
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 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)~~;   hi==n
15   {
       ...
22   }
23   //@assert lo == hi;
24   return -1;
25 }
```

```
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;
```

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`?

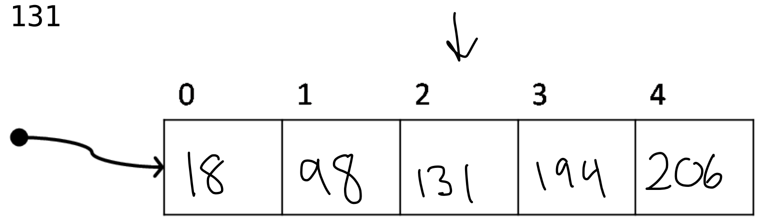
To show that this isn't always sufficient, consider an alternate loop body that performs binary search. It replaces the code at the beginning of this question.

```
{
  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;
  }
}
```

5-0/2=2 =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 = 131`
- `A =`



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

4. **Timing Code**

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.

1.5pts

4.1 In this task, consider a C0 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?

$T($ 4x $, y, z) = 16$ 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?

$T(x,$ 16 y $, z) = 16$ 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?

$T(x, y,$ 4z $) = 16$ seconds

$T(x, y, z) = C * x^2 * y * 2^2$

In practice, we rarely have a spelled out formula for the cost function T. However, we can use the above kind of calculations to experimentally investigate what this formula may be: modify one of the inputs (x, y or z) and see by how much the running time of `mystery` changes. If T is indeed $c \times x^2 \times y \times 2^z$, this is very simple as you just observed.

In the next task, we want to test the hypothesis that T is quadratic in x, but not necessarily of the simple form considered so far (for example, it may be $c \times x^2 \times y \times 2^z + k$, but we don't know that).

0.5pts

4.2 If T is indeed one of these more complex quadratic function, we can't tell by simply doubling the first argument from x to $2x$! Why?

k is constant and is unaffected by changing the first arguments

What additional test(s) can we perform to confirm (or dispel) our hunch that T is indeed some quadratic function in x?

we can test edge case extremes (i.e. run some test with x, y, and/or z equal to 0)