Lab 03: Loopty-loopty Loop

Monday January 29th

Collaboration: In lab, we encourage collaboration and discussion as you work through the problems. These activities, like recitation, are meant to get you to review what we've learned, look at problems from a different perspective and allow you to ask questions about topics you don't understand. We encourage discussing problems with other students in this lab!

Setup: Copy the lab code from our public directory to your private directory:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab03 .
% cd lab03
```

You should write your code for part (2.b) in a file, `consecutive.c0`, in the directory `lab03`. For (3.b), use `palindrome.c0` in the same directory.

Getting checked in: type `122lab` at the Linux prompt.

```
% 122lab
```

Reasoning about reversing an array of integers

You will be constructing loop invariants for a function (which we wrote for you) that reverses an array. As you define these loop invariants, take care to ensure both the **safety** of any array accesses as well as the **correctness** of the function itself.

Important! Specification functions are often inefficient, so don't use them in the actual code. Only use them inside contracts!

You will need to use the *specification function* `is_reversed` in your loop invariants. Remember, a specification function is for use in contracts to make sure the function is correct.

Here is the function header of `is_reversed`:

```
bool is_reversed(int[] first, int[] last, int i, int n)
//@requires 0 <= n && n == \length(first) && n == \length(last);
//@requires 0 <= i && i <= n;
```

This function returns `true` if the first `i` numbers in the array `first` are the reverse of the last `i` numbers of the array `last`. Now, using this specification function, fill in the loop invariants for the function `reverse` below, which reverses an array of integers.

Hint: Look at how we use `is_reversed` inside the `//@ensures` clause, and make sure that when the loop terminates, you can prove that this postcondition holds.

(1.a) Complete the missing loop invariants.

```
1  int[] reverse(int[] A, int n)
2  //@requires n == \length(A);
3  //@ensures is_reversed(A, \result, n, n);
4  {
5    int[] B = alloc_array(int, n);
6    for (int i = 0; i < n; i++)
7    //@loop_invariant ____________________________________________; // SAFETY
8    //@loop_invariant ____________________________________________; // CORRECTNESS
9    {
10     B[n - i - 1] = A[i];
11   }
12   return B;
13 }
```

1.5pt

Improving an already existing algorithm

The TAs have been tasked by Iliano to write some `c0` code which takes an array of integers and sees how many consecutive pairs of equal numbers there are in the array. They were given the following examples:

1	5	4	4	2	1	1	has 2 consecutive pairs: (4,4) and (1,1)
2	1	1	1	1	2	0	has 3 consecutive pairs: (1,1) three times
6	4	5	4	5	4	5	has 0 consecutive pairs

The best solution the TAs could come up with involved an inefficient algorithm using two **for** loops, one within the other. As you will see later on in the course, this makes their code run very slowly for large arrays. They are pretty sure it can be done with just one loop, but they need your help!

(2.a) They've put their algorithm inside a function aptly named `num_consecutive_slow(A, n)`. You can assume that it is correct (they wrote good test cases). One thing to note is that `n` can be less than the length of `A` if you just want to check the first `n` items of `A` for pairs.

Now, using `num_consecutive_slow` as a specification function, complete the loop definition for the faster algorithm in the function below, including the bounds on `i` as well as any loop invariants needed. Again, use one invariant for safety and one for correctness.

```
1  int num_consecutive_fast(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \result == num_consecutive_slow(A, n);
4  {
5    if (n == 0) return _____;
6    //@assert n >= 1;
7
8    int count = 0;
9    for (int i = _____; i < _____; i++)
10   //@loop_invariant ____________________________________________;
11   //@loop_invariant ____________________________________________;
12   {
13     ...
14   }
15   return ____________________________________________;
16 }
```

(2.b) Now open up `consecutive.c0` and fill in the loop body there. Remember, use the loop invariants to guide the code you write. When ready, test your implementation as described next.

Note: use the provided `to_arr` function to help with your testing. It takes an integer and converts it to an array of integers based on the number's digits.

```
% coin -d consecutive.c0
--> num_consecutive_fast(to_arr(1544211), 7);
2 (int)
--> num_consecutive_fast(to_arr(2111120), 7);
3 (int)
--> num_consecutive_fast(to_arr(6454545), 7);
0 (int)
```

You can test your code by running

```
cc0 -d -x consecutive.c0 test-consecutive.c0
```

3pt

Make sure your code passes this test to get credit!

Error: out of memory

Uh-oh! The 15-122 supercomputer has just run out of memory! Now how will the TAs tackle their next challenge? For some extra credit, see if you can help them out!

An **`int`** array is *palindrome* if reading it from left to right and from right to left yields the same numbers. For example, $[1, 2, 3, 2, 1]$ is palindrome, but $[1, 2, 3, 4, 2]$ is not. For years, 15-122 has been determining palindromes by checking if `A` and `reverse(A, n)` have the same contents. However, this requires allocating a new array every time you want to check if an array is palindrome. Alas, because we ran out of memory, we need a more efficient solution.

(3.a) The header for the old palindrome function is given below. It checks if `A[i,j)` is a palindrome. It uses an inclusive bound on `i`, but an exclusive bound on `j`.

```
bool is_palindrome_old(int[] A, int i, int j)
//@requires 0 <= i && i <= j && j <= \length(A);
```

The TAs have again started you off by writing the function header and the loop below. Try and fill in the loop invariants using the specification function we provided above.

Hint: you are returning `true` at the end of the function, so you want the loop invariants to help you prove that `is_palindrome_old(A, 0, n)` is true when the loop exits normally.

Remember: labs are collaborative! If this is challenging, work with your neighbors!

```
bool is_palindrome(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \result == is_palindrome_old(A, 0, n);
{
  for (int i = _____; i < _____; i++)
  //@loop_invariant ____________________________________________;
  //@loop_invariant ____________________________________________;
  {
    ...
  }
  return true;
}
```

(3.b) Again, using these loop invariants to guide you, open up `palindrome.c0` and fill in the loop body there. When ready, test your implementation as follows:

```
% coin -d palindrome.c0
--> is_palindrome(to_arr(1221), 4);
true (bool)
--> is_palindrome(to_arr(122), 3);
false (bool)
--> is_palindrome(to_arr(9), 1);
true (bool)
```

You can test your code by running

```
cc0 -d -x palindrome.c0 test-palindrome.c0
```

4pt Make sure your code passes this test to get the extra credit!