15-122: Principles of Imperative Computation — Spring 2020

Lab 4: TA Training — Thursday January 30th

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 your neighbors as you work through this lab!

Setup: This lab MUST be done in the Andrew machines. Download the handout from the website.

```
% cd private/15122
% mkdir lab04
% cd lab04
% wget https://web2.qatar.cmu.edu/~srazak/courses/15122-s20/lab/handout-04.tgz
% tar xfvz handout-04.tgz
```

Grading: For full credit, your tests should catch at least half of the bugs. For extra credit, write tests that catch all of the bugs!

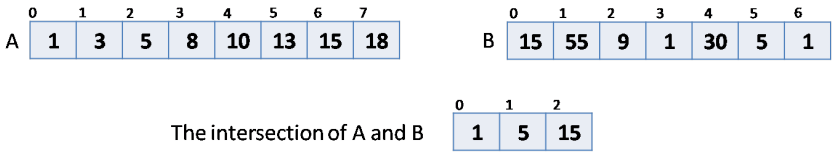
Reminder: it's okay if you don't get extra credit on every lab! The way we grade labs, you will get all the possible points as long as you attend every lab and get full credit on a handful of labs.

Introduction

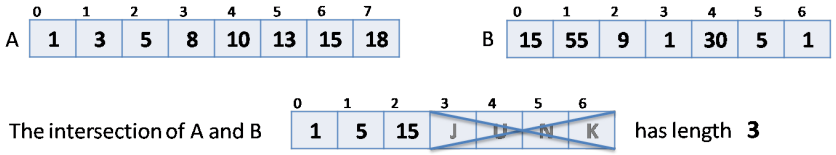
Iliano is writing a new programming assignment called sets, where he has students represent sets of integers as **int** arrays. One of the functions he wants them to write is `intersect` which computes the intersection of two arrays. The relevant section of the writeup is below:

```
int intersect(int[] A, int n, int[] B, int m, int[] intersection)
//@requires 0 <= n && n <= \length(A);
//@requires 0 <= m && m <= \length(B);
//@requires n <= \length(intersection) || m <= \length(intersection);
/*@ensures 0 <= \result && \result <= m && \result <= n; @*/ ;
```

The function `intersect` computes the *intersection* of two arrays `A` and `B`, defined as the array containing all the elements that occur in both `A` and `B` (in sorted order and without duplicates). We do **not** enforce that `A` and `B` have no duplicates nor that they be sorted. Here's an example:



Unfortunately, we cannot just return the intersection as an array and expect the client to know how long this array is, so we have to do something a little bit more fancy — we have the client give us an array that they want to be filled with the intersection, and we just return the number of integers in the intersection. The example above would now look like this:



Unfortunately, he is busy teaching 122, and so he decided to offload writing tests to his trusted TAs. Then he remembered that all his TAs are busy as well, and came up with the perfect alternative: have *students* write the tests so he can see who would be a good TA! A truly ingenious solution!

Testing code

The file `testlib.c0` contains the following helper functions, which may be useful while testing:

```
bool arr_eq(int[] A, int n, int[] B, int m)
/*@requires n <= \length(A) && m <= \length(B); @*/ ;
int[] int_array_from_string(string s);
```

(2.a) When writing test cases, we usually run the function on sample inputs and assert whether they match the output we expect. This can get quite repetitive, so we will often write a function that takes in the inputs and solution and tests to see if the actual output matches the solution. Write the following function in `set-test.c0` where, when given two arrays and the expected result, it **`assert`**s that the `intersect` function provided the correct answer

```
void run_testcase(int[] A, int a, int[] B, int b, int[] expected, int exp_len)
/*@requires 0 <= a && a <= \length(A)
         && 0 <= b && b <= \length(B)
         && 0 <= e && e <= \length(expected); @*/
```

(2.b) Inside function `run_tests` (in file `set-test.c0`), create an *exhaustive* battery of tests for `intersect`. It should return `true` when run against a correct implementation of `intersect`, and `false` when run against a buggy implementation. We will execute it against 20 different student implementations of `intersect`, some correct and many broken in different ways.

Note: You may find it it useful to organize your test file based on what you're testing for. That is, you could separate it into a "Basic Tests" section, "Tests about Duplicates" section, etc. You further can print "Basic Tests Passed!" or "Duplicates passed!" to give more information on where a problem might lie. If you like modularity (we do), these could be helper functions!

Run `./check-test`. This will run your tests on 20 student versions of `intersect`, some of which are correct implementations, and some of which are incorrect. The program `./check-test` can also be run against a specific student by calling it with `./check-test -s <student_name>` (run it first without arguments to get the student names). **Your tests must all pass on correct implementations in order to get credit**. A sample output can be found below:

```
% ./check-test
Testing student aardvark (Correct Implementation)
    Test 1... Passed
    Test 2... Failed
    Test 3... Passed
Student code failed a test (expected to pass)
...
Testing student rjsimmon (Incorrect Implementation)
    Test 1... Passed
```

```
    Test 2... Failed
    Test 3... Failed
Student code failed a test (expected to fail)... Good!
...
Tested 20 students, 9 students had no failed tests, 11 students had failed tests.
(No credit to be awarded --- your code fails students with correct code)
```

1.5pt **(2.c)** Your `run_tests` returns `true` on all correct implementations of `intersect`.

3pt **(2.d)** Additionally, your `run_tests` returns `false` on half the buggy implementations of `intersect`.

4pt **(2.e)** Additionally, your `run_tests` returns `false` on all the buggy implementations of `intersect`.