15-122: Principles of Imperative Computation — Spring 2024

Lab 04: TA Training — Tuesday 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 other students in this lab!

Setup: Copy the lab code from our public directory to your private directory in your unix.qatar.cmu.edu machine:

```
% cd private/15122
% cp -R /afs/andrew/course/15/122/misc/lab04 .
% cd lab04
```

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.

Submission: . To submit, create a tar file by executing this command:

```
% tar cfzv handin.tgz set-test.c0
```

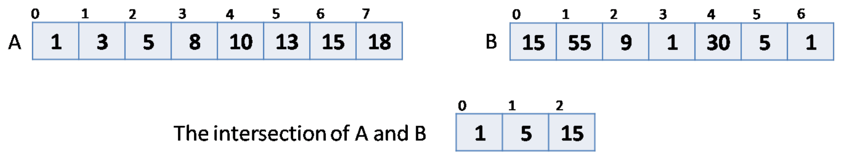
and submit it to autolab, under the lab name.

Introduction

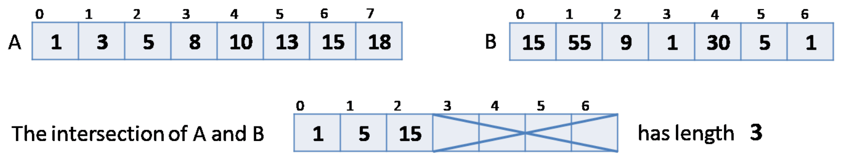
Iliano is writing a new programming assignment called sets, where he has students represent sets of integers as `int` arrays. One of the functions he wants them to write is `intersect` which computes the intersection of two arrays. The relevant section of the writeup is below:

```
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:



(the contents of the returned array from index 3 on is unimportant).

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 function, which may be useful while testing:

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

(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 — especially when working with C0 arrays. Instead, we will write a function that takes in the inputs and the expected output to see if the solution matches.

Complete the following function in `set-test.c0`. Its inputs are strings consisting of space-delimited integers that we convert for you into the two input arrays of `intersect` and their expected intersection.

```
void run_testcase(string A_as_str, string B_as_str, string Expected_as_str);
```

There are exactly two blanks you need to fill in. The rest of the function has been written for you. You don't need to understand the provided code, but the TAs will be happy to explain it to you if you are curious!

Here is how you would call this function on the example given earlier:

```
run_testcase("1 3 5 8 10 13 15 18", "15 55 9 1 30 5 1", "1 5 15");
```

(2.b) Inside function `run_tests` (in file `set-test.c0`), create an *exhaustive* battery of tests for `intersect`. We will execute it against 22 different student implementations of `intersect`. Just **two** are correct, while the rest are 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 22 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 22 students, 9 students had no failed tests, 13 students had failed tests.
(No credit to be awarded --- your code fails students with correct code)
```

1.5pt **(2.c)** Your `run_tests` reports no failure on all correct implementations of `intersect`.

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

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