15-122: Principles of Imperative Computation

Lab Week 4 — Rob Simmons and Tom Cortina

Unless arranged *well in advance* with an instructor, you may *only* attend the lab you're registered for.

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

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

Grading: For 2 points, complete task (1.a). For 3 points, finish at least task (2.a). Note that for (2.a) and (2.b), the most important part of the problems is the testing of the functions, not the functions themselves.

Timing

On the Linux machines, there is a way to determine the actual running time of a program. You use the `time` command followed by the program name (and its arguments) that you want to time. For example, to time an executable `a.out` in your current directory, you would enter:

```
time ./a.out
```

and you would get an output that looks something like this, that shows that the program used 4.602 seconds of user time:

```
Timing 1000 times with 2^18 elements
0
4.602u 0.015s 0:04.63 99.5% 0+0k 0+0io 0pf+0w
```

NOTE: When timing code, do not use `-d` during compilation. The extra debugging done by contracts will increase the overall runtime and can affect the overall asymptotic complexity of the algorithm you are timing.

(1.a) In this activity, there are six programs, `timingtest1, timingtest2, ..., timingtest6` that allow you to vary the input size for the algorithm they're running by giving an optional `-n` argument. The default size, 1000, is a good starting place for all 6 programs. For example, to time the first program using the input size 1000, you would enter: `time timingtest1 -n 1000`

Your job is to determine the asymptotic complexity (runtime) of all six programs expressed using big O notation as a function of n, in its simplest, tightest form. HINT: Exactly one of them is $O(logn)$. Do this by running the programs for varying sizes of n and plotting your results (or looking for obvious patterns). Work with your neighbors to get done faster!

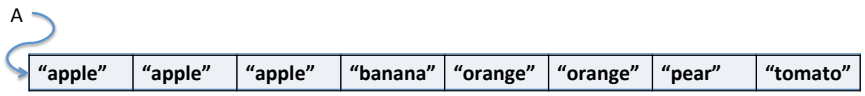
Testing

In these activities, we will have you work on some functions and build comprehensive test cases for them. When you write functions, you should write your own test cases to test for correctness and safety, especially with edge (exceptional) cases, before you submit to Autolab. You need to get in the habit of testing your own code and not using Autolab as your compiler to collect points. If you test your code well, the points will follow.

(2.a) Implement a function matching the following declaration in the file `duplicates.c0`.:

```
int count_distinct(string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
//@ensures 0 <= \result && \result <= n;
```

where `n` represents the size of the subarray of `A` that we are considering. This function should return the number of distinct strings in the array (i.e., if the same value occurs more than once it should contribute only one to the count). There are five distinct fruits in this array:



so calling `count_distinct(A,8)` should return 5.

Your implementation should have a linear ($O(n)$) asymptotic running time. Think carefully about this: how can you take advantage of the preconditions to achieve this?

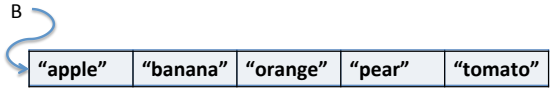
Once you have a function for testing, complete a `main` function in the file `duplicates-test.c0` that tests your `count_distinct` function fully. This file has a few simple examples to get you started. Work with your neighbors to determine a full set of tests that should be implemented. Think about edge cases (e.g. arrays of size 0, size 1), arrays with all one string, arrays with no duplicates, arrays with pairs or triples of duplicates, etc. You won't get credit for this part unless your TA feels you've tested the function thoroughly (and all tests pass, of course).

Next, you will take a sorted array of strings and return a new sorted array that contains the same strings without duplicates; i.e., the new array will contain every string in the original one, but all the strings will be distinct. The length of the new array should be just big enough to hold the resulting strings. The code for this exercise should be put in the file `duplicates.c0` and your additional unit tests should be put in the file `duplicates-test.c0`.

(2.b) Implement a function matching the following function declaration:

```
string[] remove_duplicates(string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
```

where `n` represents the size of the subarray of `A` that we are considering. The strings in the array should be sorted before the array is passed to your function. This function should return a new array that contains only one copy of each distinct string in the array `A`, and your new array should be sorted as well. Calling `string[] B = remove_duplicates(A,8)` should give you this array:



Just like `count_distinct`, your implementation should have a **linear** asymptotic running time. **Your solution should include annotations for at least 2 strong postconditions.** To get credit for this exercise, you will need to demonstrate comprehensive testing of your function to the TA.