15-122: Principles of Imperative Computation, Fall 2014
Lab 4: Processing Arrays of Strings

Tom Cortina(tcortina@cs) and Rob Simmons(rjsimmon@cs)

Monday, September 22, 2014

You are expected to complete at least the first two exercises for full lab credit. Remember that in lab, you are allowed to collaborate!

SETUP: This lab uses the same `arrayutil.c0` library that is distributed with the `doslingos` programming assignment. This library uses strings as its base type instead of integers. To make compiling easier, store your `c0` files for this lab in your `doslingos` directory. If you haven't downloaded the starter code for `doslingos`, now is a great time to do that!

1 Searching for Duplicates

In this programming exercise, you will take a sorted array of strings and determine if all of the strings are distinct and count how many distinct strings are in the array. The code for this exercise should be put in a file `duplicates.c0`. You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. You may include additional annotations for assertions as necessary. You may include any auxiliary functions you need in the same file, but you should not include a `main()` function. To test your code using `coin`:

```
coin -d lib/*.c0 duplicates.c0
```

Once you test your code some, you can use Autolab for further testing.

Exercise 1. Implement a function matching the following function declaration:

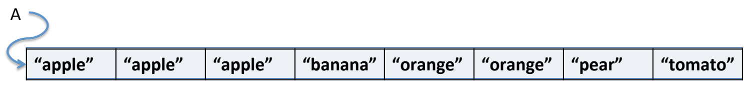
```
bool all_distinct(string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
```

where `n` represents the size of the subarray of `A` that we are considering. This function should return `true` if the given string array contains no repeated strings and `false` otherwise.

Exercise 2. Implement a function matching the following function declaration:

```
int count_distinct(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. 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** asymptotic running time. Think carefully about this: how can you take advantage of the preconditions to reduce the running time of the function?

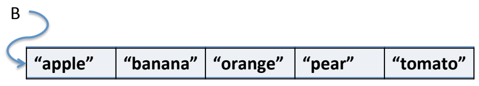
2 Removing Duplicates

In this programming exercise, 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 tests should be put in the file `duplicates-test.c0`.

Exercise 3. 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 3 strong post-conditions.**