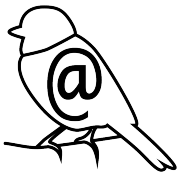
15-122: Principles of Imperative Computation, Fall 2014

Programming 4: Twitter



Due: Wednesday, October 8, 6pm

This week we will do some exercises on searching and sorting arrays of integers and strings. You may want to refer to Section 8 in the C0 language reference, Section 2.2 of the C0 library reference, the page on Strings in the C0 Tutorial (`http://c0.typesafety.net/tutorial/Strings.html`).

1 Getting the code

The code handout for this assignment is available from the course web page. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

To download and unzip the code, do the following:

1.1 Mac OS

1. On the assignments page of the course web site, right-click the Programming 3 `Code` link, and then click "Save Link as...", and choose to save it in `Documents/comp211/`.

2. Start a new terminal window and do the following:

   ```
   cd comp211
   tar -xzvkf hw4-handout.tgz
   ```

   This will make a directory called `comp211/hw4-handout` and extract all of the code for the assignment.

3. The file `hw4-handout/README.txt` contains a description of the various files, and instructions on running them.

1.2 Windows

1. On the assignments page of the course web site, right-click the Programming 3 `Code` link, and then click "Save Link as...", and choose to save it in your `comp211/`. To find this directory in your web browser, look in

   ```
   C:\cygwin\home\YOURNAME\comp211\
   ```

2. Start a new terminal window and do the following:

```
cd comp211
tar -xzkvf hw4-handout.tgz
```

   This will make a directory called `comp211/hw4-handout` and extract all of the code for the assignment.

3. The file `hw4-handout/README.txt` contains a description of the various files, and instructions on running them.

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`.

2.1 All Distinct

Task 1 (2 pt) *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);
```

This function is given an array `A` *and a number* `n` *and considers the segment of the array from* `0` *to* `n` *(exclusive of* `n`*). This function should return* `true` *if the given array segment contains no repeated strings and* `false` *otherwise. The function assumes that the given array is sorted.*

Note that for strings, being sorted means being in "dictionary" (lexicographic) order—strings are compared character by character from left to right, and the first character where they differ determines the order (e.g. `"ab"` $<$ `"ac"`), or if one is a prefix of the other, then that one is smaller (e.g. `"a"` $<$ `"ac"`). Individual characters are ordered by their ASCII codes.

You can test whether two strings are equal using the `string_equal` function.

You should test this function by loading the code in `coin`

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

and then running the `test_all_distinct()` function.

Task 2 (1pt) *Add at least three more tests to* `test_all_distinct`*, which check that it is correct in the following circumstances:*

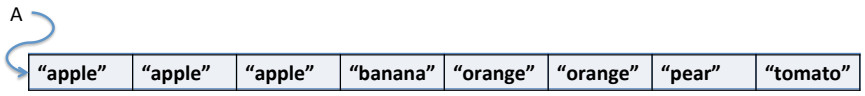
- *When the only duplicate is the first element of the array.*
- *When the only duplicate is the last element of the array.*
- *When there are duplicates in the array but beyond* `n`.

2.2 Count Distinct

Task 3 (2 pt) *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);
```

This function is given an array `A` *and a number* `n` *and considers the segment of the array from* `0` *to* `n` *(exclusive of* `n`*). 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). For example, there are five distinct fruits in this array:*



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

Your implementation should have a ***linear*** *running time*[1]*. Think carefully about this: what precondition should you exploit to reduce the running time of the function?*

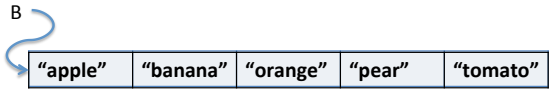
Task 4 (1 pt) *Write a function* `void test_count_distinct()` *with some tests for* `count_distinct`.

2.3 Remove Duplicates

Task 5 (4 pts) *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);
```

Here `n` represents the size of the subarray of `A` that we are considering—the result should consider only the strings in `A` from position 0 (inclusive) to `n` (exclusive). The strings in `A` 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` (from position `0` (inclusive) to `n` (exclusive)), and your new array should be sorted as well. Calling `string[] B = remove_duplicates(A,8)` should give you this array:



[1] assuming `string_equal` takes a constant amount of time

Just like `count_distinct`, your implementation should have a **linear** running time (assuming string equality takes constant time).

Task 6 (2pt) *Your solution should include annotations for at least 3 strong postconditions, and should include enough loop invariants that you could prove the postconditions from your loop invariants, and prove that all array accesses are in bounds.*

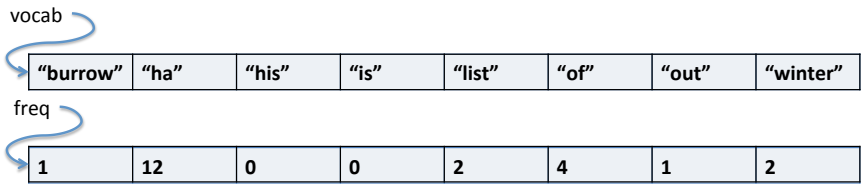
Task 7 (1pt) *Write a* `void test_remove_duplicates()` *function.*

3 What's the frequency, Kenneth?

The story: You're doing a linguistics study on how Twitter is affecting people's use of language. For this study, you want to figure out how frequently various words are tweeted. That is, you want to compute a frequency table, which lists, for various words, how often they have been tweeted in the tweets you analyzed:

```
burrow 1
ha 12
his 0
is 0
list 2
of 4
out 1
winter 2
```

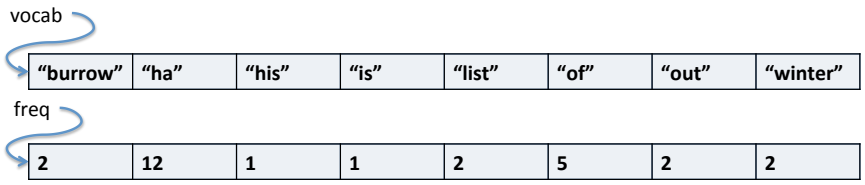
We represent a frequency table by two arrays `vocab` and `freq` of the same length `v`. `vocab` is an array of strings and `freq` is an array of integers. For each number `0 <= i < v`, `vocab[i]` is some string ("winter"), and `freq[i]` is the corresponding frequency (2). This can be depicted as follows:



We additionally maintain the invariant that `vocab` is *sorted*. This way, the two arrays can be thought of as a *dictionary*: The sorted words stored in `vocab` are *keys* and the frequency counts stored in `freq` are *values*. Given a word, we can look up its frequency according to the dictionary by searching for it in `vocab`, which gives its index `i`, and then retreiving the frequency at position `i` in `freq`.

In this exercise, you will write a function for updating a frequency table by processing some tweets. The function `count_vocab` that you will write updates the frequency counts based on the words in some Twitter data. For example, consider a Twitter corpus containing only this tweet by weatherman Scott Harbaugh:

Running `count_vocab` on the above frequency table would update the frequency table to the following:



because `count_vocab` should update the counts of all of the words that are in the frequency table that occur in the new data. Moreover, `count_vocab` should return the number of words that are *not* in the frequency table, so that you can see how many words you're missing. In this case, `count_vocab` would return 1 because only one word, "Phil," is not in our example vocabulary.

In general, the input Twitter corpus will be represented by an array `new_words` (note that `new_words` is *not* sorted!), containing the words in the tweets.

You are given string search algorithms in `lib/stringsearch.c0`, which implement linear search, as described in Lecture 9, as well as binary search (the divide in half search we talked about in the first lecture). We will discuss the code for binary search in class next week, but to do this assignment you don't need to understand the code for it; you just need to know that it meets the following specification:

```
// Linear search
int linsearch(string x, string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
        || ((0 <= \result && \result < n)
            && string_equal(A[\result], x)); @*/

// Binary search
int binsearch(string x, string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
/*@ensures (-1 == \result && !is_in(x, A, 0, n))
        || ((0 <= \result && \result < n)
            && string_equal(A[\result], x)); @*/
```

Task 8 (6 pts) *In the file* `count.c0`*, write a function definition* `count_vocab` *that matches the following function declaration:*

```
int count_vocab(string[] vocab, int[] freq, int table_length,
                string[] tweet_words, int tweet_words_length,
                bool fast)
//@requires table_length == \length(vocab) && table_length == \length(freq);
//@requires \length(tweet_words) == tweet_words_length;
//@requires is_sorted(vocab, 0, table_length) && all_distinct(vocab, table_length);
```

The function should return the number of occurrences of words in `tweet_words` *that do* not *appear in the array* `vocab`*, and should change the frequency counts in* `freq` *with the number of*

times each word in the vocabulary appears. If a word appears multiple times in `tweet_words`*, you should count each occurrence separately, so the tweet "ha ha ha LOL LOL" would cause the the frequency count for "ha" to be incremented by 3 and would cause 2 to be returned, assuming LOL was not in the vocabulary.*

Note that a precondition of `count_vocab` is that `vocab` is sorted, a fact you should exploit. Your function should use the linear search algorithm when `fast` is false and it should use the binary search algorithm when `fast` is true. You can implement this choice with a simple if statement that decides which function to call – duplicating a lot of code is unnecessary and unhelpful.

See `README.txt` for instructions on compiling and running `count.c0`. This instructions include the functions `linsearch` and `binsearch` so you can use them in your code. You should test this task by running it on the data files described below.

Data Analysis

Your data: There are some data files for you to analyze in the `texts/` directory. You can use these to test `count_vocab`.

The following two files contain sorted lists of vocabulary words:

- `vocab_sorted_small.txt` - A small sorted list of vocabulary words.
- `vocab_sorted.txt` - A large sorted list of vocabulary words.

You can use these to make an initial frequency table where all of the frequencies are 0.

The following files contain tweets.

- `twitter_1.txt` - Scott Harbaugh's tweet above.
- `twitter_1k.txt` - A small collection of 1000 tweets to be used for testing slower algorithms.
- `twitter_200k.txt` - A larger collection of 200k tweets to be used for testing faster algorithms.

The file `README.txt` explains how to compile and run your code on these files.

Task 9 (1 pt) *In a comment at the end of* `count.c0`*, list the top 100 most tweeted words in* `twitter_200k.txt` *(using the vocabulary in* `vocab_sorted.txt`*), along with their frequencies. Explain what happens if you try to calculate this top 100 list using linear search instead of binary search.*