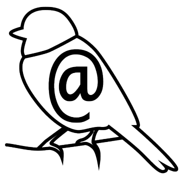
15-122: Principles of Imperative Computation, Spring 2014

Homework 3 Programming: DosLingos



Due: Monday, February 10, 2014 by 22:00

This week we will do three short exercises on searching and sorting arrays of integers and strings. We haven't talked too much about strings yet: 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`), or Appendix A of this writeup for more about strings.

The code handout for this assignment is at

`http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/prog3.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in.

There is no limit on the number of times you may hand in this assignment on Autolab. For tasks 1-4, we will provide some feedback on your test cases – test cases that are very good at finding bugs will be recorded on the scoreboard and treated as a bonus points activity. The test cases for tasks 1-4 are technically optional, however: these test cases will not contribute to your grade. (The test cases you write in Task 5 will be graded.)

1 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 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.

Your tests should be put in a file `duplicates-test.c0` that *does* have a `main()` function. The autograder will test your tests, but these tests will not be graded.

Task 1 (1 pt) *Implement a function matching the following function declaration:*

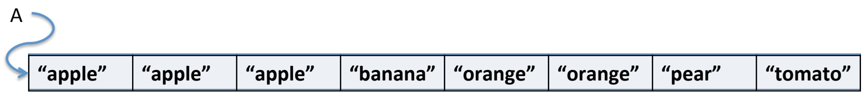
```
bool all_distinct(string[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A, 0, n);
```

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.*

Task 2 (1 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);
```

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:*



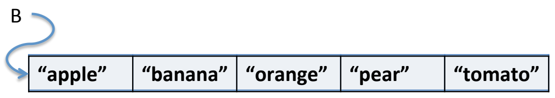
so calling `count_distinct(A,8)` *should return* 5.

Your implementation should have a ***linear*** *asymptotic running time. Think carefully about this: what precondition should you exploit to reduce the running time of the function? (A question for you to think about: given that it could be met, why don't we impose this requirement on your implementation of* `all_distinct`*?)*

Task 3 (3 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);
```

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:*

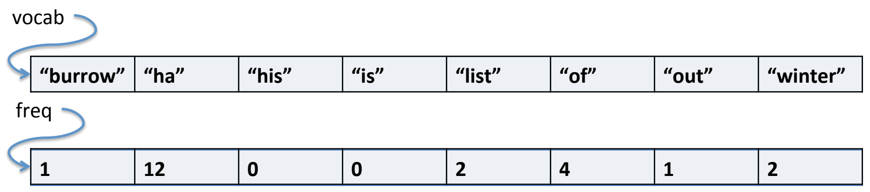


Just like `count_distinct`*, your implementation should have a* ***linear*** *asymptotic running time. Your solution should include annotations for at least 3 strong postconditions.*

2 DosLingos (Counting Common Words)

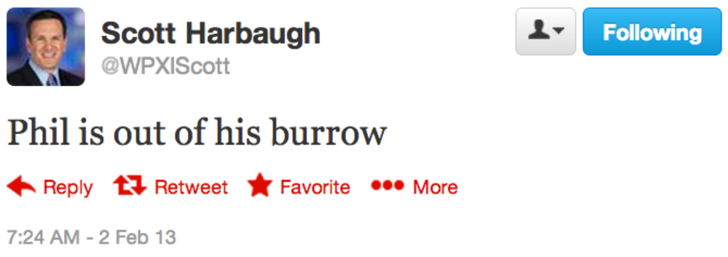
The story: You're working for a Natural Language Processing (NLP) startup company called DosLingos.[1] Already, your company has managed to convince thousands of users to translate material from English to Spanish for free. In a recent experiment, you had users translate only newswire text and you've managed to train your users to recognize words in an English newspaper. However, now you're considering having these same users translate Twitter tweets as well, but you're not sure how many words of English Twitter dialect your Spanish-speaking users will be able to recognize.

Your job: In this exercise, you will write a function for analyzing the number of tokens from a Twitter feed that appear (or not) in a user's vocabulary. The user's expected vocabulary will be represented by a sorted array of strings `vocab` that has length `v`, and we will maintain another integer array, `freq`, where `freq[i]` represents the number of times we have seen `vocab[i]` in tweets so far (where $i \in [0, v)$).



This is an important pattern, and one that we will see repeatedly throughout the semester in 15-122: the (sorted) vocabulary words stored in `vocab` are *keys* and the frequency counts stored in `freq` are *values*.

The function `count_vocab` that we will write updates the values – the frequency counts – based on the unsorted Twitter data we are getting in. For example, consider a Twitter corpus containing only this tweet by local weatherman Scott Harbaugh:

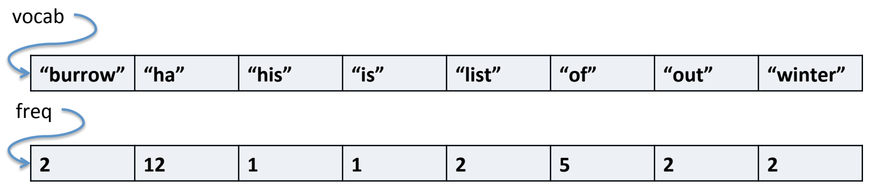


Scott Harbaugh
@WPXIScott

Phil is out of his burrow

7:24 AM - 2 Feb 13

We would expect `count_vocab(vocab,freq,8,"texts/scottweet.txt",b)` to return 1 (because only one word, "Phil," is not in our example vocabulary), leave the contents of `vocab` unchanged, and update the frequency counts in `freq` as follows:



[1] Any resemblance between this scenario and Dr. Luis von Ahn's company DuoLingo (www.duolingo.com) are purely coincidental and should not be construed otherwise.

Your data: DosLingos has given you 4 data files for your project in the `texts/` directory:

- `news_vocab_sorted.txt` - A sorted list of vocabulary words from news text that DosLingos users are familiar with.
- `scotttweet.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.

Your tools: DosLingos already has a C0 library for reading text files, provided to you as `lib/readfile.c0`, which defines a type `string_bundle` and implements the following functions:

```
// first call read_words to read in the content of the file
string_bundle read_words(string filename)
```

You need not understand anything about the type `string_bundle` other than that you can extract its underlying `string` array and the length of that array:

```
// to determine the length of the array in the string_bundle, use:
int string_bundle_length(string_bundle sb)

// access the array inside of the string_bundle using:
string[] string_bundle_array(string_bundle sb)
//@ensures \length(\result) == string_bundle_length(wl);
```

Here's an example of these functions being used on Scott Harbaugh's tweet:

```
$ coin lib/readfile.c0
--> string_bundle bund = read_words("texts/scotttweet.txt");
bund is 0xFFAFB8E0 (struct fat_string_array*)
--> string_bundle_length(bund);
6 (int)
--> string[] tweet = string_bundle_array(bund);
tweet is 0xFFAFB670 (string[] with 6 elements)
--> tweet[0];
"phil" (string)
--> tweet[5];
"burrow" (string)
```

Being connoisseurs of efficient algorithms, DosLingos has also implemented their own set of string search algorithms in `lib/stringsearch.c0`, which you may also find useful for this assignment:

```
// 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)); @*/
```

The code for this exercise should be put in a file `doslingos.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. You can include functions from the `lib/readfile.c0` and `lib/stringsearch.c0` libraries in your code: the compilation instructions given in `README.txt` include these libraries.

Your tests should be put in a file `doslingos-test.c0` that has a `main()` function. The autograder will test your tests, but these tests will not be graded.

Task 4 (5 pts) *Create a file* `doslingos.c0` *containing a function definition* `count_vocab` *that matches the following function declaration:*

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

The function should return the number of occurrences of words in the file `tweetfile` *that do not appear in the array* `vocab`*, and should update the frequency counts in* `freq` *with the number of times each word in the vocabulary appears. If a word appears multiple times in the* `tweetfile`*, 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. It should not be an error if this addition causes overflow.*

Note that a precondition of `count_vocab` *is that the* `vocab` *must be sorted, a fact you should exploit. Your function should use the linear search algorithm when* `fast` *is set to 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.

3 Unit testing

DosLingos's old selection sort is no longer up to the task of sorting large texts to make vocabularies. Your colleagues currently use two sorts, both given in `sort.c0`:

```
void sort(string[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);

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

The first is an in-place sort like we discussed in class, and the second is a copying sort that must leave the original array `A` unchanged and return a sorted array of length `upper-lower`. (Note that neither of these conditions are directly expressed in the contracts.)

DosLingos decided to pay another company to write faster sorting algorithms with the same interface. Unfortunately, they didn't realize that the other company was a closed-source shop, so now your company's future is depending on code you can't see – you know that the contracts are set up correctly, but you don't know anything about the implementation. This causes (at least) two big problems.

First, you can't prove that your sorting functions always respect their contracts – the best you can do is give a counterexample, writing a test that causes the `@ensures` statement to fail. If the outside contractors give you this completely bogus implementation...

```
void sort(string[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(A, lower, upper);
{
  return;
}
```

...then you can show that their implementation is buggy by writing a test file with a `main()` function that performs a sort and observing that the `@ensures` statement fails when you compile the test with `-d` and run it.

Second, even code that *does* satisfy the contracts may not actually be correct! For example, this `sortcopy` function will never fail the postcondition, but it is definitely incorrect:

```
string[] sortcopy(string[] A, int lower, int upper)
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
//@ensures is_sorted(\result, 0, \length(\result));
{
  return alloc_array(string, upper-lower);
}
```

In order to catch this kind of incorrect sort, you will have to write a test file with a `main()` function that runs the sort and then uses assertions to check that other correctness conditions hold – to catch this bug, the `is_in()` and/or `binsearch()` functions might be helpful, for instance, though they certainly aren't necessary. You might end up preferring `//@assert ...;` to `assert(...);` in this assignment, as the former lets you use `\length(A)` to check the length of an array.

To get full credit on the next task, you'll need to write tests with extra assertions that will fail with assertion errors both if the outside contractors wrote a sometimes-postcondition-failing implementation and if they exploited the contracts to give you a bogus-but-contract-abiding implementation.

Task 5 (5 pts) *Write two files,* `sort-test.c0` *and* `sortcopy-test.c0`*, that test the two sorting functions. The autograder will assign you a grade based on the ability of your unit tests to pass when given a correct sort and fail when given various buggy sorts. Your tests must still be safe: it should not be possible for your code to make an array access out-of-bounds when -d is turned on.*

You do not need to catch all our bugs to get full points, but catching additional tests will be reflected on the scoreboard (and considered for bonus points).

Because you cannot access all of our buggy implementations except via the autograder, your grade on this task *will* be given as soon as you hand in your work.

3.1 Testing your tests

You can test your functions with DosLingos's own (correct) selection sort algorithm, and on the two awful badly broken implementations given in this section, by running the following commands:

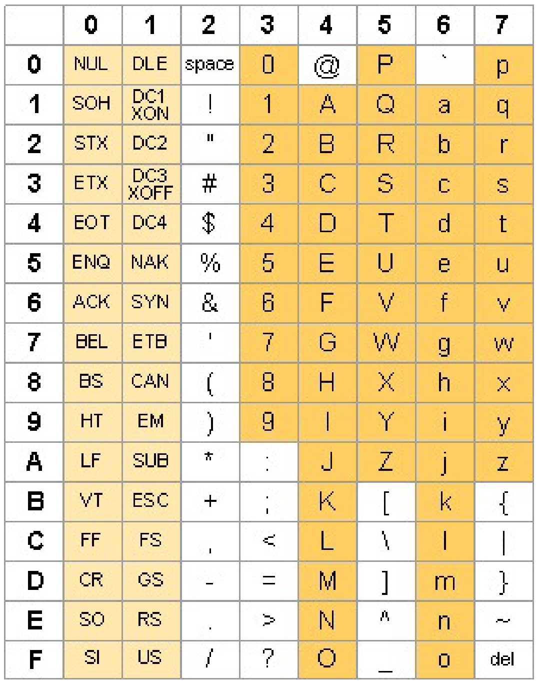
```
% cc0 -d -w lib/*.c0 sort.c0 sort-test.c0
% ./a.out
% cc0 -d -w lib/*.c0 sort.c0 sortcopy-test.c0
% ./a.out
% cc0 -d -w lib/*.c0 sort-awful.c0 sort-test.c0
% ./a.out
% cc0 -d -w lib/*.c0 sortcopy-awful.c0 sortcopy-test.c0
% ./a.out
```

All four of these tests should compile and run, but the last two invocations of `./a.out` should trigger a contract violation if your tests are more than minimal. We will only test one function at a time, so `sort-test.c0` must only reference the `sort()` function and `sortcopy-test.c0` must only reference the `sortcopy()` function. Both can reference the specification function `is_sorted` and all other functions defined in `lib/arrayutil.c0` and `lib/stringsearch.c0`.

A String Processing Overview

In the C0 language, a `string` is a sequence of characters. Unlike languages like C, a string is not the same as an array of characters. There is a library of string functions (which you include in your code by `#use <string>`) that you can use to process strings:

```
// Returns the length of the given string
int string_length(string s)

// Returns the character at the given index of the string.
// If the index is out of range, aborts.
char string_charat(string s, int idx)
//@requires 0 <= idx && idx < string_length(s);

// Returns a new string that is the result of concatenating b to a.
string string_join(string a, string b)
//@ensures string_length(\result) == string_length(a) + string_length(b);

// Returns the substring composed of the characters of s beginning at
// index given by start and continuing up to but not including the
// index  given by end.  If end <= start, the empty string is returned
string string_sub(string a, int start, int end)
//@requires 0 <= start && start <= end && end <= string_length(a);
//@ensures string_length(\result) == end - start;

bool string_equal(string a, string b)

int string_compare(string a, string b)
//@ensures -1 <= \result && \result <= 1;

// Create strings from ints
string string_fromint(int i)

// Create strings from bools
string string_frombool(bool b)

// Create strings from chars
string string_fromchar(char c)
//@requires c != '\0';
//@ensures string_length(\result) == 1;
```



	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Figure 1: The ASCII table

The `string_compare` function performs a *lexicographic* comparison of two strings, which is essentially the ordering used in a dictionary, but with character comparisons being based on the characters' ASCII codes, not just alphabetical. For this reason, the ordering used here is sometimes whimsically referred to as "ASCIIbetical" order. A table of all the ASCII codes is shown in Figure 1. The ASCII value for `'0'` is 0x30 (48 in decimal), the ASCII code for `'A'` is 0x41 (65 in decimal) and the ASCII code for `'a'` is 0x61 (97 in decimal). Note that ASCII codes are set up so the character `'A'` is "less than" the character `'B'` which is less than the character `'C'` and so on, so the "ASCIIbetical" order coincides roughly with ordinary alphabetical order.