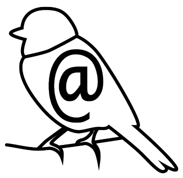
15-122: Principles of Imperative Computation, Spring 2013

Homework 2 Programming: Twitterlab



Due: Monday, February 11, 2013 by 23:59

For the programming portion of this week's homework, you'll write two files C0 files corresponding to two different string processing tasks, and a two other files that performs unit tests on a potentially buggy sorting implementation:

- `duplicates.c0` (described in Section 2)
- `count_vocab.c0` (described in Section 3)
- `sort-test.c0` (described in Section 4)
- `sort_copy-test.c0` (described in Section 4)

You should submit these files electronically by 11:59 pm on the due date. Detailed submission instructions can be found below.

Assignment: String Processing (20 points)

Starter code. Download the file `hw2-handout.tgz` from the course website. When you unpack it, you will find a `lib/` directory with several C0 files, including `stringsearch.c0` and `readfile.c0`. You will also see a `texts/` directory with some sample text files you may use to test your code. You should not modify or submit code in the `lib` directory.

For this homework, you are not provided any `main()` functions. Instead, you should write your own `main()` functions for testing your code. You should put this test code in separate files from the ones you will submit for the problems below (e.g. `duplicates-test.c0`). You may hand in these files or not.

Compiling and running. You will compile and run your code using the standard C0 tools. For example, if you've completed the program `duplicates` that relies on functions defined in `stringsearch.c0` and you've implemented some test code in `duplicates-test.c0`, you might compile with a command like the following:

```
% cc0 duplicates.c0 duplicates-test.c0
```

Don't forget to include the `-d` switch if you'd like to enable dynamic annotation checking, but this check should be turned off when you are evaluating the running time of a function.

Submitting. Once you've completed some files, you can submit them to Autolab. There are two ways to do this:

From the terminal on Andrew Linux (via cluster or ssh) type:

```
% handin hw2 duplicates.c0 count_vocab.c0 \
    sort-test.c0 sort_copy-test.c0 README.txt
```

Your score will then be available on the Autolab website.

Your files can also be submitted to the web interface of Autolab. To do so, please `tar` them, for example:

```
% tar -czvf sol.tgz duplicates.c0 count_vocab.c0 \
    sort-test.c0 sort_copy-test.c0 README.txt
```

Then, go to `https://autolab.cs.cmu.edu` to submit.

You can submit this assignment as often as you would like. When we grade your assignment, we will consider the most recent version submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. For this assignment, we have provided the pre- and postconditions for many of the functions that you will need to implement. However, you should provide loop invariants and any assertions that you use to check your reasoning. If you write any "helper" functions, include precise and appropriate pre- and postconditions.

You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Unit testing. You should write unit tests for your code. This involves writing a separate `main()` function that runs individual functions many times with various inputs, asserting that the expected output is produced. You should specifically choose function inputs that are tricky or are otherwise prone to fail. While you will not directly receive a large amount of credit for these tests, your tests will help you check the correctness of your code, pinpoint the location of bugs, and save you hours of frustration.

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. If you find yourself writing the same code over and over, you should write a separate function to handle that computation and call it whenever you need it. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on Piazza if you're unsure of what constitutes good style.

Task 0 (5 points) 5 points on this assignment will be given for style.

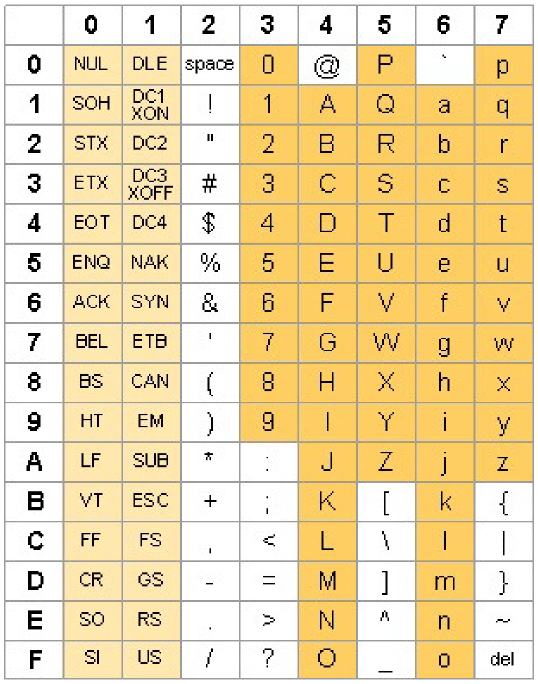
1 String Processing Overview

The three short programming problems you have for this assignment deal with processing strings. 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. (See section 8 in the C0 language reference, section 2.2 of the C0 library reference, and the page on Strings in the C0 tutorial[1] for more information on strings). 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;
```

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.

[1] `http://c0.typesafety.net/tutorial/Strings.html`



	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

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. The length of the new array should be just big enough to hold the resulting strings. Place your code for this section in a file called `duplicates.c0`; you'll want this file to start with `#use "lib/stringsearch.c0"` in order to get the `is_sorted` function from class adapted to string arrays. Implement unit tests for all of the functions in this section in a file called `duplicates_test.c0`.

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

```
bool is_unique(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_unique(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 unique strings in the array, and your implementation should have an appropriate asymptotic running time given the precondition.*

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);
```

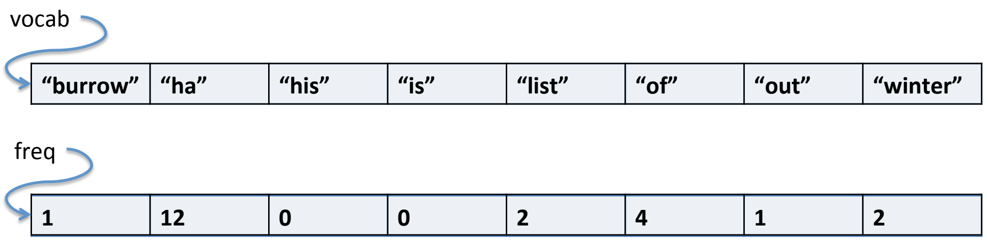
where `n` *represents the size of the subarray of* `A` *that we are considering. The strings in the array should be sorted before the array is passed to your function. This function should return a new array that contains only one copy of each distinct string in the array* `A`*. Your new array should be sorted as well. Your implementation should have a **linear** asymptotic running time. Your solution should include annotations for at least 3 strong postconditions.*

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.

3 DosLingos (Counting Common Words)

The story: You're working for a Natural Language Processing (NLP) startup company called DosLingos.[2] 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.

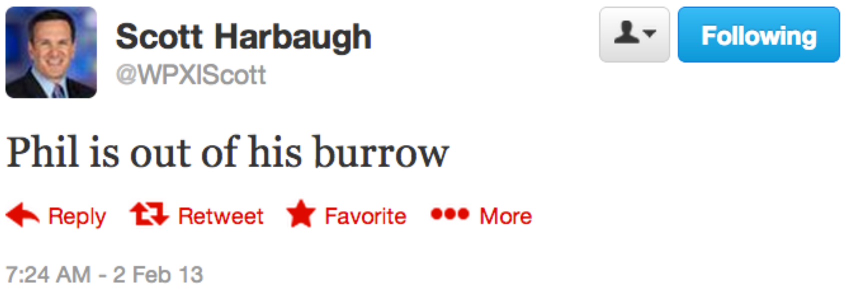
Your job: In this exercise, you will write a functions for analyzing the number of tokens from a Twitter feed that appear (or not) in a user's vocabulary. The user's expected vocabulary will be represented by a sorted array of strings `vocab` that has length `v`, and we will maintain another integer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



[2]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.

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*.

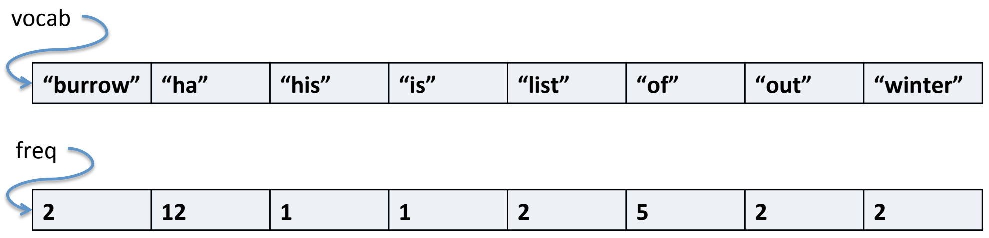
The function `count_vocab` that we will write updates the values – the frequency counts – based on the unsorted Twitter data we are getting in. For example, consider a Twitter corpus containing only this tweet by weatherman Scott Harbaugh:



Scott Harbaugh
@WPXIScott

Phil is out of his burrow

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:



"burrow"	"ha"	"his"	"is"	"list"	"of"	"out"	"winter"
2	12	1	1	2	5	2	2

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 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:

```
// access the array inside of the string_bundle using:
string[] string_bundle_array(string_bundle sb)

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

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:

```
int linsearch(string x, string[] A, int n) // Linear search
  //@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)); @*/

int binsearch(string x, string[] A, int n) // Binary search
  //@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)); @*/
```

You can include these libraries in your code by writing `#use "lib/readfile.c0"` and `#use "lib/stringsearch.c0"`.

Task 4 (4 pts) *Create a file* `count_vocab.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) && is_unique(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.*

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.*

Because `count_vocab` uses the `is_unique` function you wrote earlier, when you write a `count_vocab-test.c0` function to test your implementation, you'll want to include the file `duplicates.c0` on the command line:

```
% cc0 duplicates.c0 count_vocab.c0 count_vocab-test.c0
```

Task 5 (1 pt) *Create a file* `README.txt` *answering the following questions:*

1. *Give the asymptotic running time of* `count_vocab` *under (1) linear and (2) binary search using big-O notation. This should be in terms of* `v`*, the size of the vocabulary, and* `n`*, the number of tweets in* `tweetfile`*.*

2. *How many seconds did it take your function to run on the linear search strategy (*`fast`*=false) using the small 1K twitter text?* **Do not use contract checking via the** `-d` **option.** *Also, these tests should use cc0,* not *Coin, so you'll need to write a file* `count_vocab-time.c0` *to help you when you do this step.*

   You should use the Unix command `time` *in this step. You can report either wall clock time or CPU time, but say which one you used.*

   Example: `time ./count_vocab`

3. *How many seconds did it take for the binary search strategy (*`fast`*=true) to run on the small 1K twitter text?*

4. *How many seconds did it take for the binary search strategy (*`fast`*=true) on the larger 200K twitter text?*

Submit this file along with the rest of your code.

4 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 `lib/stringsort.c0`:

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

string[] sort_copy(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 `sort_copy` function will never fail the postcondition, but it is definitely incorrect:

```
string[] sort_copy(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 `//@assert` statements 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.

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 6 (5 pts) *Write two files,* `sort-test.c0` *and* `sort_copy-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).

4.1 Testing your tests

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

```
% cc0 -d lib/stringsort.c0 sort-test.c0
% ./a.out
% cc0 -d lib/stringsort.c0 sort_copy-test.c0
% ./a.out
% cc0 -d lib/sort-awful.c0 sort-test.c0
% ./a.out
% cc0 -d lib/sort_copy-awful.c0 sort_copy-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 `sort_copy-test.c0` must only reference the `sort_copy()` function. Both can reference the specification function `is_sorted` and all other functions defined in `lib/stringsearch.c0`. You can `#use "lib/stringsearch.c0"` in your tests, but it is important that you do **not** `#use "lib/stringsort.c0"`.