15-122: Principles of Imperative Computation, Spring 2017

Programming Homework 4: DosLingos

Due: Thursday 16th February, 2017 by 9pm

This week we will do some relatively small exercises centered around searching and sorting arrays of integers and strings. We compared characters and strings for equality during the puzzle hunt portion of our first programming assignment; Appendix A of this writeup talks a little big more about string comparison, which is necessary when we think about sorted arrays of strings.

The code handout for this assignment is on Autolab and at

`http://cs.cmu.edu/~15122/hw/doslingos-handout.tgz`

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

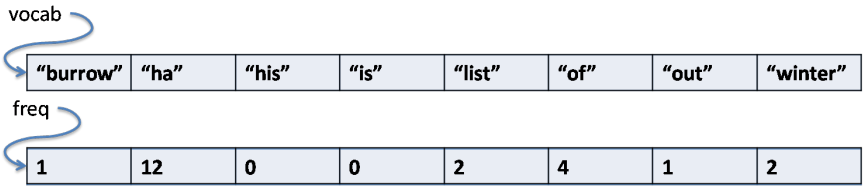
The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a FIVE (5) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty. **Be aware that only Task 3 will be graded by Autolab when you hand in your work. You should examine Autolab's output to make sure the other tasks compile.** ***If you don't check Autolab's outputs and there are compilation errors, you may end up receiving no credit for the assignment.***

The other tasks will be autograded or graded by hand after the assignment deadline. You will need to use the test cases you write for task 3, contracts, and deliberate programming to ensure correctness of the other tasks.

1 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 newswire text and you've managed to train your users to recognize words in an English newspaper. Now you're considering having these same users translate Shakespeare, and you're not sure how many words 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 text corpus (like the complete works of Shakespeare) 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 the text corpus so far (where $i \in [0, v)$).

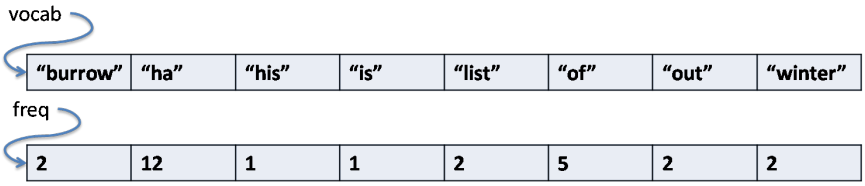


This is an important pattern, and one that we will see repeatedly throughout the semester in 15-122: the (sorted) vocabulary words stored in `vocab` are *keys* and the frequency counts stored in `freq` are *values*.

The function `count_vocab` that we will write updates the values — the frequency counts — based on the text corpus we are analyzing. As an example, consider a text made up not from Shakespeare but from this tweet by local weatherman Scott Harbaugh:

We would expect `count_vocab(vocab,freq,8,"texts/scott_tweet.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 (`http://www.duolingo.com`) are *purely* coincidental.



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.
- `scott_tweet.txt` - Scott Harbaugh's tweet above.
- `sonnets.txt` - A small test file: 122 of Shakespeare's sonnets.
- `shakespeare.txt` - A larger test file: the complete works of Shakespeare.

You can write more data files of your own!

Your tools: DosLingos already has a C0 library for reading text files, converting all letters to lowercase, and separating out words, provided to you as `lib/readfile.c0`, which defines a type `bundle_t` and implements the following functions:

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

You need not understand anything about the type `bundle_t` 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(bundle_t sb)

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

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

```
% coin -d lib/readfile.c0
--> bundle_t B = read_words("texts/scott_tweet.txt");
B is 0x7047F0 (struct string_bundle_header*)
--> string_bundle_length(B);
6 (int)
--> string[] tweet = string_bundle_array(B);
tweet is 0x704B60 (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.

Task 1 (4 points) Add to `doslingos.c0` a definition of the function `count_vocab`:

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

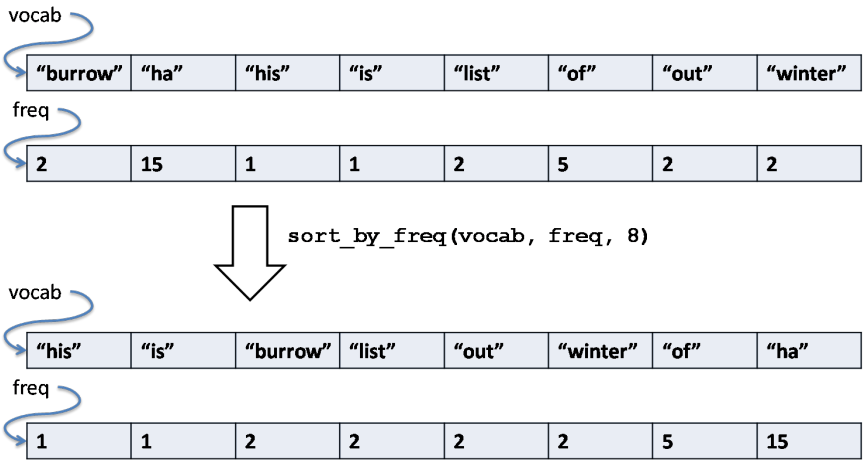
The function should return the number of occurrences of words in the file `corpus` that do not appear in the array `vocab`, and should update the frequency counts in `freq` with the number of times each word in the vocabulary appears. If a word appears multiple times in the `corpus`, you should count each occurrence separately, so a file containing "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. The easiest thing to do is just increment the frequency counts without regard to overflow, and you should do that.)

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.*

The precondition `all_distinct(vocab, b)` enforces that there are no duplicate words. You DO NOT have to write this function or include this part of the precondition; we promise not to test your code on any sorted vocabularies with duplicate words. However, you may write and include an `all_distinct` precondition if you want to. If you write it, include it in the same file `doslingos.c0`.

2 Sorting by Frequency

To really utilize our frequency counts, it is useful to be able to take our vocabulary list and frequency counts and re-arrange both arrays so that the frequency counts are sorted.



You'll want to adapt one of the sorting functions we discussed in lecture for this task (in other words, it's okay to start with the lecture code and modify it). You'll need to modify the lecture code so that it sorts the two arrays *together*. In the example above, when we move 12 from index 1 to index 7 in `freq`, we also have to move "ha" from index 1 to index 7 in `vocab`. In principle, it's not too hard to adapt an integer array sorting algorithm to an algorithm that sorts two arrays: whenever you swap two elements in the array you're sorting (`freq`), make sure the same swap is performed on the other array (`vocab`).

For full credit on this task, you will need to give a sorting algorithm that is **fast** and **stable**. By fast, we mean that it will be $O(n \log n)$ on the inputs we will give it. By stable, we mean that the relative positions of equivalent elements (words with the same frequencies) in the inputs are preserved in the output. The quicksort algorithm we discussed in class would meet the **fast** criteria: while it is $O(n^2)$ in the worst case, this behavior won't be triggered by our frequency counts in our text corpus. However, the quicksort algorithm from lecture **isn't stable**. If you straightforwardly adapt quicksort, the words "burrow", "list", "out", and "winter" are likely to end up in a different order, resulting in an output that doesn't match the example above. (Selection sort is neither fast nor stable.)

The easiest way to make a fast, stable sort is probably to modify mergesort, which we talked about briefly in class; code for mergesort is published alongside the lecture notes on divide-and-conquer sorting. You're also welcome to try and implement a stable `partition` function, which will make quicksort stable.

Task 2 (7 points) Add to `doslingos.c0` a definition of the function `sort_by_freq`:

```
void sort_by_freq(string[] vocab, int[] freq, int v)
//@requires v == \length(vocab) && v == \length(freq);
//@ensures is_sorted_int(freq, 0, v);
```

You can adapt any code from lecture, but cite the source in a comment. Remember that the `arrayutil.c0` you're using for this assignment is for dealing with string arrays, not integer arrays, so most lecture code won't compile at first. It's okay to remove contracts that depend on the lecture version of `arrayutil.c0` from your sort, but at least leave enough contracts to reason about the safety of your code.

3 Unit testing

The functions you wrote in the first two tasks could fail in many ways. On certain inputs, they might fail internal assertions or postconditions (contract failures), and on other inputs they might happily return invalid results (contract exploits).

Task 3 (8 points) Write a file, `doslingos-test.c0`, that tests your implementation of your first two tasks. The autograder will assign you a grade based on the ability of your unit tests to pass when given a correct implementation and fail when given various buggy implementations. 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.

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. We'll run tests with contracts (`-d`) enabled, so the largest text files should not be used in your unit tests.

You may find it useful to use the functions provided in C0's `parse` library. These functions provide a convenient way of creating arrays with specific contents. It's not necessary to use this library to test your code, but you may find that writing

```
string[] A = parse_tokens("I love 15-122");
```

is more convenient than writing

```
string[] A = alloc_array(string, 3);
A[0] = "I";
A[1] = "love";
A[2] = "15-122";
```

3.1 Testing your tests

You can test your functions with your own implementation, and with an awful and badly broken implementation, by running the following commands:

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

Both tests should compile and run, but the last invocation of `./a.out` should trigger a assertion to fail if your tests are more than minimal. *Even if your test cases fail on the awful implementation, they still might not be particularly useful test cases.*

4 Analyzing the results

Once you've carefully tested your `doslingos.c0` implementations, you have a powerful set of tools for analyzing your text corpus. For the last part of this assignment, you'll run such an analysis. *It is a violation of the academic integrity policy of this course to compare the answers in this section with other students.*

Task 4 (6 points) Create a file `analysis.c0` containing a `main()` function. We will compile and run this file as follows:

```
% cc0 -w -o analysis lib/*.c0 doslingos.c0 analysis.c0
% ./analysis texts/news_vocab_sorted.txt texts/shakespeare.txt
```

Where the first argument to the function is the filename for the (sorted) dictionary and the second argument is the text corpus. See `echo.c0` for an example of how to handle command-line arguments in C0. Note that, to receive credit, your implementation shall work for an arbitrary dictionary and an arbitrary corpus, passed as parameters as above.

Your analysis should use the sorted word list to compute and print out human-readable answers to the following questions:

- What are the four most common in-dictionary words in the text corpus, and what are their frequencies? (Ties, if they occur, can be broken in any way you want.)
- What in-dictionary words appear exactly 128 times in the text corpus?
- How many in-dictionary words appear exactly once in the text corpus? (Don't print them all out, just give the number.)
- How many distinct frequencies (zero counts as a frequency) are there for the in-dictionary words in text corpus? (In the example on the top of page 2, the answer would be 5, because 0, 1, 2, 4, and 12 are the distinct frequencies.)

Your code only needs to work on reasonable inputs. Specifically, don't worry about checking that the first argument is a file that is actually sorted. If you use the tools you developed in this assignment correctly, you should be able to compute the answers from scratch in a couple of seconds. We don't need the output of your analysis to obey a strict format, but here's the rough format you should follow:

```
The four most common words in the text corpus are:
  1. ___REDACTED___ (appears ___REDACTED___ times)
  2. ___REDACTED___ (appears ___REDACTED___ times)
  3. ___REDACTED___ (appears ___REDACTED___ times)
  4. ___REDACTED___ (appears ___REDACTED___ times)

These words appeared 128 times in the text corpus: ___REDACTED___

There are ___REDACTED___ singletons in the corpus.

There are ___REDACTED___ distinct frequency counts in the corpus.
```

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. One of the functions in the string library (which you include in your code by `#use <string>`) is `string_compare`:

```
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. We can convert between the **char** type and the integer ASCII codes with the `char_ord(c)` and `char_chr(i)` functions, also available in the string library. 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.

	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