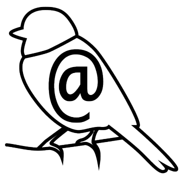
15-122: Principles of Imperative Computation, Spring 2024

Programming Homework 4: Speller



Due: Thursday 8th February, 2024 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 B of this writeup talks a little big more about string comparison, which is necessary when we think about sorted arrays of strings.

Download the assignment handout from the course website or Autolab. 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 (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

Be aware that only Task 7 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 7, contracts, and deliberate programming to ensure correctness of the other tasks.

1 Introduction

The story: The Hackers from the Lost Web, a cyber-criminal organization that specializes in ransomware, has disabled all spellchecking software on campus. You have a term paper due and you are worried your professor will lower your grade if there are spelling mistakes in your paper. Being a 15-122 student, you decide to write your own spellchecker with the help of the friendly 15-122 staff.

Your tools: Your diligent 15-122 TAs have given you a C0 library for reading text files, converting all letters to lowercase, and separating out words. It is provided to you as `lib/readfile.o0`, which defines a type `bundle_t` and implements the following functions:

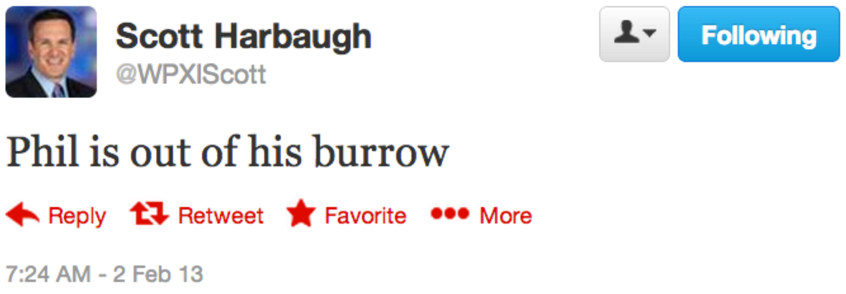
```
// First call read_words to read in the content of the file
bundle_t read_words(string filename)
```

You don't need to 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 a tweet by local weatherman Scott Harbaugh:



Phil is out of his burrow

7:24 AM - 2 Feb 13

```
% coin -d lib/readfile.o0
--> 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)
```

Your data: You are given 2 dictionaries and 3 text files for your project in the `texts/` directory:

- `dict.txt` — A standard dictionary you will be using.
- `small-dict.txt` — A minuscule dictionary.
- `scott-tweet.txt` — A tiny text file: Scott Harbaugh's tweet.
- `sloth.txt` — Another small text file: the sloth example below.
- `shakespeare.txt` — A larger text file: the complete works of Shakespeare.

You can write more data files of your own!

2 Spellchecking a Word

In this exercise, you will write a binary search function for spellchecking a word against a dictionary. The dictionary is represented as a sorted array `dict` of strings and has length `d`, while the word is represented as a string `w`. The function `check_word` that you will write returns a boolean value indicating whether `w` is in the dictionary.

For example, we expect that `check_word(dict, d, "sloth")` returns `true` while `check_word(dict, d, "sloathe")` returns `false` with respect to the standard dictionary (file `dict.txt`).

The code for this exercise should be put in a file `speller.c0`. You must include annotations for the precondition(s), postcondition(s) and loop invariant(s) for each function. Include additional annotations for assertions as necessary. You may put any auxiliary functions you need in the same file, but you should *not* include a `main()` function. You can use functions from the `lib/readfile.o0` file in your code.

Task 1 (2 points) In file `speller.c0`, implement the function `check_word`:

```
bool check_word(string[] dict, int d, string w)
//@requires \length(dict) == d;
//@requires is_sorted(dict, 0, d) && no_dupes(dict, 0, d);
```

This function should return whether the word `w` is in the dictionary `dict`. The preconditions let you assume that the dictionary `dict` is sorted and has no duplicates, a fact you should exploit for your code to run in time $O(\log d)$ — an implementation based on linear search is likely to fail subsequent tasks.

3 Spellchecking a Text

Now that we are able to spellcheck a word, let's spellcheck a whole text. Consider this text:

```
Sloths are named after the capital sin of sloth; however, their usual
idleness is due to metabolic adaptations for conserving energie.
Aside from their surprising speed during emergency flights from
predators, other notable traits of sloths include their strong body
and their ability to host symbiotic algee on their furs.
```

3.1 Word by Word

An obvious way to spellcheck this text is to first call `check_word(dict, d, "sloths")` and then call `check_word(dict, d, "are")` and so on. Let's automate this idea by storing the text in an array of strings, one word in each location, and calling `check_word` inside a loop that iterates over each word. Rather than just returning whether the whole text was or wasn't correctly spelled, our spelling functions will return an array containing all the misspelled words in the text.

Task 2 (2 points) In `speller.c0`, complete the definition of the function

```
int check_text_naive(string[] dict, int d, string[] text, int t, string[] miss)
//@requires \length(dict) == d;
//@requires \length(text) == t;
//@requires \length(miss) >= t;
//@requires is_sorted(dict, 0, d) && no_dupes(dict, 0, d);
//@ensures 0 <= \result && \result <= t;
//@ensures no_dupes(miss, 0, \result);
```

The function takes in five arguments. The first two are the dictionary (which is sorted and has no duplicates) and its length. The next two are the text you are spellchecking and its length. The final argument, `miss`, is the array that you will use to return the misspelled words you find. This array needs to be at least as big as the text array because every word might be misspelled. As you find misspelled words, you will write each of them in order into the array `miss`, starting at index 0, without repetitions. The function returns the number of words you wrote into `miss`, i.e., the number of unique misspelled words in the text. Later on in this course, we will learn more elegant ways of returning two pieces of data from a function. For example, for the text given earlier, the `miss` array should contain the strings `idlleness`, `energie` and `algee` (in that order in the first three slots) and `\result` should be equal to 3.

3.2 Spellchecking Sorted Texts without Duplicates

The complexity of `check_text_naive` is at least $O(t \log d)$ because it calls `check_word` (which has cost $O(\log d)$) on each of the t words in the text.[1] You can easily see from our example that we will end up spellchecking common words like `sloths`, `are` and `their` multiple times. A text without duplicates is clearly immune from this problem. The cost is still $O(t \log d)$ however. *Can we do better?*

The 15-122 TAs unanimously claim ***we can*** if the text is *sorted*! They give you the challenge to implement a spellchecker for sorted texts without duplicates that runs in time $O(d + t)$ — and doesn't use `check_word`.

[1] It's actually even worse given that the `miss` array shall not contain duplicates.

Task 3 (4 points) In `speller.c0`, complete the definition of the function

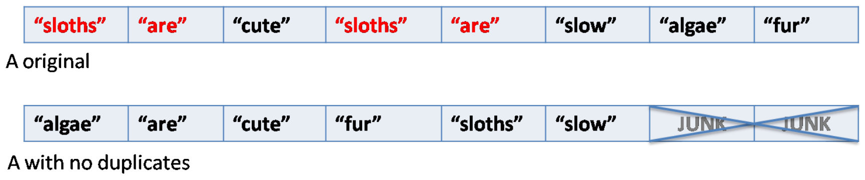
```
int check_sorted_text(string[] dict, int d, string[] text, int t, string[] miss)
//@requires \length(dict) == d;
//@requires \length(text) >= t;
//@requires \length(miss) >= t && t >= 0;
//@requires is_sorted(dict, 0, d) && no_dupes(dict, 0, d);
//@requires is_sorted(text, 0, t) && no_dupes(text, 0, t);
//@ensures 0 <= \result && \result <= t;
//@ensures is_sorted(miss, 0, \result) && no_dupes(miss, 0, \result);
```

You should spellcheck only the first `t` words of `text` (why will become clear later). As before, the function should return the number of misspelled words, and you should populate the array `miss` with those words. Since the text is sorted, so will the computed misspelled words. Make sure that the running time of this function is indeed $O(d + t)$ as it may time out on Autolab otherwise.

3.3 Sorting a Text and Removing Duplicates

But everyday texts like essays and email are not sorted and they do contain duplicates. Our next task will be to take such a text and generate another text that is sorted and contains the same words, only once.

One convenient way to remove duplicates is while sorting the text array. For example:



As you can see, the array `A` has length 8, but there are two duplicates in the array (highlighted in red). If we sort and remove the duplicates, we get an array that is still of size 8, but only the first 6 slots have values we actually care about.

For full credit on the next two tasks, you will need to write a sorting algorithm that is **fast** and **removes duplicates**. By fast, we mean that it will be $O(t \log t)$ on a t-word input text. The easiest way to make a fast, duplicate-removing sort is to modify mergesort, which we talked about in class; code for mergesort is published alongside the lecture notes on divide-and-conquer sorting. We advise you to look into this code and to work out some examples before you move on to modifying mergesort. Keep in mind that `alloc_array(type, n)` has cost $O(n)$, so be careful about how large the arrays you allocate are, as this might impact the time running time of your implementation.

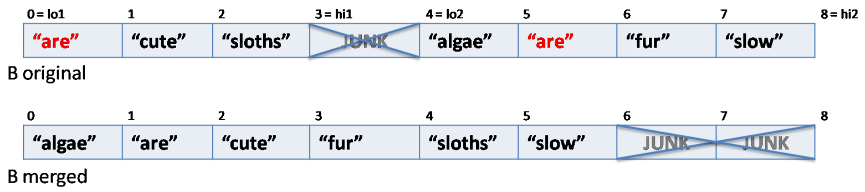
In order to cleanly remove duplicates and keep track of the number of unique elements in the array, we have rewritten the prototypes of mergesort as follows:

```
int merge(string[] A, int lo1, int hi1, int lo2, int hi2)
//@requires 0 <= lo1 && lo1 < hi1 && hi1 <= lo2 && lo2 < hi2 && hi2 <= \length(A);
//@requires is_sorted(A, lo1, hi1) && no_dupes(A, lo1, hi1);
//@requires is_sorted(A, lo2, hi2) && no_dupes(A, lo2, hi2);
//@ensures 0 <= \result && \result <= hi2 - lo1;
//@ensures is_sorted(A, lo1, lo1 + \result) && no_dupes(A, lo1, lo1 + \result);
```

```
int mergesort(string[] A, int lo, int hi)
//@requires 0 <= lo && lo <= hi && hi <= \length(A);
//@ensures 0 <= \result && \result <= hi - lo;
//@ensures is_sorted(A, lo, lo + \result) && no_dupes(A, lo, lo + \result);
```

These functions now return an **int** instead of **void** because we will use this value to keep track of the number of unique elements in the resulting array. The arguments to `mergesort` are the same as those in the standard mergesort implementation. However, you will notice that our `merge` function takes four array indices as arguments instead of three. Instead of passing it `lo`, `mid` and `hi`, we pass it two sorted intervals `[lo1, hi1)` and `[lo2, hi2)` that contain no duplicates that we need to merge together. We can't use a standard `merge` because it would merge the junk at the end of each segment into the output array.

For example, after we call `merge(B, 0, 3, 4, 8)` on the array `B` pictured below, the array `B` will contain the merged and sorted intervals, and return 6.



Your next job is to first implement `merge` and then `mergesort`. Make sure you test your mergesort implementation to ensure it is removing duplicates.

Task 4 (5 points) Add to `speller.c0` a definition of the function `merge` as defined above.

Task 5 (3 points) Add to `speller.c0` a definition of the function `mergesort` as defined above.

3.4 Better Spellchecking

With a way to sort and remove duplicates from a text and a way to quickly spellcheck a text with these characteristics, we can write a spellchecker for a generic text that, in many cases, runs faster than `check_text_naive`.

Task 6 (2 points) Add to `speller.c0` a definition of the function `check_text_better` that first sorts a text and then spellchecks it. Write its asymptotic complexity in a comment.

```
int check_text_better(string[] dict, int d, string[] text, int t, string[] miss)
//@requires \length(dict) == d;
//@requires \length(text) == t;
//@requires \length(miss) >= t;
//@requires is_sorted(dict, 0, d) && no_dupes(dict, 0, d);
//@ensures 0 <= \result && \result <= t;
//@ensures  is_sorted(miss, 0, \result) && no_dupes(miss, 0, \result);
```

As always, the function should return the number of misspelled words, and you should populate the array `miss` with those words, in alphabetical order.

4 Unit Testing

The functions you wrote in the earlier 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 7 (4 points) In file `speller-test.c0`, write test cases that test your implementation of your `check_word` and `check_text_better`. Your `speller-test.c0` should test **only** these two functions. Do not assume the existence of `merge`, `mergesort`, `check_text_naive`, or `check_sorted_text`. 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.

You are welcome to hand in additional data files if you wish, but do not hand in data files given to you as part of the handout.

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";
```

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/*.o0 speller.c0 speller-test.c0
% ./a.out
% cc0 -d -w lib/*.o0 speller-awful.c0 speller-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.*

5 Analyzing the Results

Once you've carefully tested your `speller.c0` implementations, you have a powerful set of tools for analyzing your input text. 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 8 (3 points) Create a file `analysis.c0` containing a `main()` function. Here's how to compile and two examples of how to run the compiled program:

```
% cc0 -w -o analysis lib/*.o0 speller.c0 analysis.c0
% ./analysis texts/small-dict.txt texts/sloth.txt
% ./analysis texts/dict.txt texts/shakespeare.txt
```

The first command-line argument to the compiled program `./analysis` is the filename for the (sorted) dictionary and the second argument is the text. 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 text — possibly much larger than the above example — passed as parameters as above.

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

- How many unique misspelled words are there in the text (relative to the dictionary)?
- How many unique misspelled words of length exactly 8 are there in the text relative to the dictionary?
- Consider the unique misspelled words in the text relative to the dictionary, sorted alphabetically. List the first 4 such words of length 15.
- How many times does the alphabetically-last misspelled word (this would be "idlleness" in the sloth example) appear in the text? If the text does not contain misspelled words, point this out.

Your code only needs to work on reasonable inputs. Specifically, don't worry about checking that the first argument is a file of words that are 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:

```
There are ___REDACTED___ misspelled words in the text.

There are ___REDACTED___ misspelled words of length 8 in the text.

Here are the first 4 misspelled words of length 15 in the text:
 1. ___REDACTED___
 2. ___REDACTED___
 3. ___REDACTED___
 4. ___REDACTED___

The alphabetically-last misspelled word in the text is ___REDACTED___
and appears ___REDACTED___ times.
```

If a requested word doesn't exist, print `(N/A)`.

Your output shall not contain any debug printouts.

A REFERENCE: Library Interfaces

A.1 `readfile`

```
/**************************** Interface ****************************/

// typedef _______* bundle_t;

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

// to determine the length of the array in the bundle_t, use:
int string_bundle_length(bundle_t wl);

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

You can also display this interface by running the terminal command

```
% cc0 -i lib/readfile.o0
```

A.2 `arrayutil`

This version of the `arrayutil.c0` library operates on array of strings, and is otherwise identical to what we saw in class. You can display its interface with the command:

```
% cc0 -i lib/arrayutil.o0
```

B Appendix: String Processing Overview

In C0, a **string** is a sequence of characters (but it is not an array of characters). One of the functions in the string library (which you include in your code by **#use** <string>) is string_compare:

```
/* \result  < 0 if a is less than b,
           == 0 if a is equal to b, and
            > 0 if a is greater than b
*/
int string_compare(string a, string b) ;
```

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. 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.

You can find more information about the string library functions at https://c0.cs.cmu.edu/docs/c0-libraries.pdf.

Partial ASCII Table

32	20	␣	64	40	@	96	60	‘
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_			

Figure 1: The ASCII table