Problem Set 5: Mispellings

This is CS50. Harvard University. Fall 2014.

Table of Contents

Objectives

- Allow you to design and implement your own data structure.
- Optimize your code's (real-world) running time.

Recommended Reading

- Pages 18 – 20, 27 – 30, 33, 36, and 37 of http://www.howstuffworks.com/c.htm.
- Chapter 26 of *Absolute Beginner's Guide to C*.
- Chapter 17 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.

- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you’ve written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code online so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course’s own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set’s problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set’s problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff’s solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course’s own lessons and integrate into your own work, even while respecting this policy’s other constraints.
- Giving or showing to a classmate a solution to a problem set’s problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual’s work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz’s questions or answers prior to taking the quiz.

- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Ready

First, join Jackson and Lauren for tours of singly linked lists and hash tables.

http://www.youtube.com/watch?v=5nsKtQuT6E8
Next, join Kevin for a tour of tries.

http://www.youtube.com/watch?v=NKr6gWcXkIM
Finally, remind yourself how `valgrind` works if you've forgotten or not yet used!

http://www.youtube.com/watch?v=fvTsFjDuag8

Getting Started

Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

```
update50
```

to ensure that your appliance is up-to-date!

Like Problem Set 4, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/Dropbox
```

in order to navigate to your `~/Dropbox` directory. Then execute

```
wget http://cdn.cs50.net/2014/fall/psets/5/pset5/pset5.zip
```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called `pset5.zip` in your `~/Dropbox` directory. Unzip it by executing the below.

```
unzip pset5.zip
```

If you again execute

```
ls
```

you should see that you now also have a `pset5` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f pset5.zip
```

Now dive into that `pset5` directory by executing the below.

```
cd pset5
```

Now execute

```
ls
```

and you should see that the directory contains the below.

```
dictionary.c  dictionary.h  Makefile  questions.txt  speller.c
```

Interesting!

Anyhow, theoretically, on input of size *n*, an algorithm with a running time of *n* is asymptotically equivalent, in terms of *O*, to an algorithm with a running time of 2*n*. In the real world, though, the fact of the matter is that the latter feels twice as slow as the former.

The challenge ahead of you is to implement the fastest spell-checker you can! By "fastest," though, we're talking actual, real-world, noticeable time—none of that asymptotic stuff this time.

In `speller.c`, we've put together a program that's designed to spell-check a file after loading a dictionary of words from disk into memory. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you!

Before we walk you through `speller.c`, go ahead and open up `dictionary.h` with `gedit`. Declared in that file are four functions; take note of what each should do. Now open up `dictionary.c`. Notice that we've implemented those four functions, but only barely, just enough for this code to compile. Your job for this problem set is to re-implement those functions as cleverly as possible so that this spell-checker works as advertised. And fast!

Let's get you started.

Makefile

Recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. However, as your programs grow in size, make won't be able to infer from context anymore how to compile your code; you'll need to start telling make how to compile your program, particularly when they involve multiple source (i.e., `.c`) files, as in the case of this problem set. And so we'll utilize a `Makefile`, a configuration file that tells make exactly what to do. Open up `Makefile` with `gedit`, and let's take a tour of its lines.

The line below defines a variable called `CC` that specifies that make should use `clang` for compiling.

```
CC = clang
```

The line below defines a variable called `CFLAGS` that specifies, in turn, that `clang` should use some flags, most of which should look familiar.

```
CFLAGS = -ggdb3 -O0 -Qunused-arguments -std=c99 -Wall -Werror
```

The line below defines a variable called `EXE`, the value of which will be our program's name.

```
EXE = speller
```

The line below defines a variable called `HDRS`, the value of which is a space-separated list of header files used by `speller`.

```
HDRS = dictionary.h
```

The line below defines a variable called `LIBS`, the value of which is should be a space-separated list of libraries, each of which should be prefixed with `-l`. (Recall our use of `-lcs50` earlier this term.) Odds are you won't need to enumerate any libraries for this problem set, but we've included the variable just in case.

```
LIBS =
```

The line below defines a variable called `SRCS`, the value of which is a space-separated list of C files that will collectively implement speller.

```
SRCS = speller.c dictionary.c
```

The line below defines a variable called `OBJS`, the value of which is identical to that of `SRCS`, except that each file's extension is not `.c` but `.o`.

```
OBJS = $(SRCS:.c=.o)
```

The lines below define a "target" using these variables that tells make how to compile speller.

```
$(EXE): $(OBJS) Makefile
    $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
```

The line below specifies that our `.o` files all "depend on" `dictionary.h` and `Makefile` so that changes to either induce recompilation of the former when you run `make`.

```
$(OBJS): $(HDRS) Makefile
```

Finally, the lines below define another target for cleaning up this problem set’s directory.

```
clean:
    rm -f core $(EXE) *.o
```

Know that you’re welcome to modify this `Makefile` as you see fit. In fact, you should if you create any `.c` or `.h` files of your own. But be sure not to change any tabs (i.e., `\t`) to spaces, since make expects the former to be present below each target. To be safe, uncheck **Use Spaces** under **Tab Width** at the bottom of `gedit`'s window before modifying `Makefile`.

The net effect of all these lines is that you can compile `speller` with a single command, even though it comprises quite a few files:

```
make speller
```

Even better, you can also just execute:

```
make
```

And if you ever want to delete speller plus any `core` or `.o` files, you can do so with a single command:

```
make clean
```

In general, though, anytime you want to compile your code for this problem set, it should suffice to run:

```
make
```

speller.c

Okay, next open up `speller.c` with `gedit` and spend some time looking over the code and comments therein. You won't need to change anything in this file, but you should understand it nonetheless. Notice how, by way of `getrusage`, we'll be "benchmarking" (i.e., timing the execution of) your implementations of `check`, `load`, `size`, and `unload`. Also notice how we go about passing `check`, word by word, the contents of some file to be spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics.

Notice, incidentally, that we have defined the usage of `speller` to be

```
Usage: speller [dictionary] text
```

where `dictionary` is assumed to be a file containing a list of lowercase words, one per line, and `text` is a file to be spell-checked. As the brackets suggest, provision of `dictionary` is optional; if this argument is omitted, `speller` will use `/home/cs50/pset5/dictionaries/large` by default. In other words, running

```
./speller text
```

will be equivalent to running

```
./speller ~cs50/pset5/dictionaries/large text
```

where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, `speller` will not be able to load any dictionaries until you implement `load` in `dictionary.c`! Until then, you'll see **Could not load**.)

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size, as with `gedit`. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with `\n`). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide `speller` with a `dictionary` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In `/home/cs50/pset5/dictionaries/small` is one such dictionary. To use it, execute

```
./speller ~cs50/pset5/dictionaries/small text
```

where `text` is the file you wish to spell-check. Don't move on until you're sure you understand how `speller` itself works!

Odds are, you didn't spend enough time looking over `speller.c`. Go back one square and walk yourself through it again!

questions.txt

Okay, technically that last problem induced an infinite loop. But we'll assume you broke out of it. Open up `questions.txt` with `gedit` and answer each of the following questions in one or more sentences.

0. What is pneumonoultramicroscopicsilicovolcanoconiosis?
1. According to its `man` page, what does `getrusage` do?
2. Per that same man page, how many members are in a variable of type `struct rusage`?
3. Why do you think we pass `before` and `after` by reference (instead of by value) to `calculate`, even though we're not changing their contents?
4. Explain as precisely as possible, in a paragraph or more, how `main` goes about reading words from a file. In other words, convince us that you indeed understand how that function's `for` loop works.
5. Why do you think we used `fgetc` to read each word's characters one at a time rather than use `fscanf` with a format string like `"%s"` to read whole words at a time? Put another way, what problems might arise by relying on `fscanf` alone?

6. Why do you think we declared the parameters for `check` and `load` as `const`?

texts

So that you can test your implementation of `speller`, we've also provided you with a whole bunch of texts, among them the script from *Austin Powers: International Man of Mystery*, a sound bite from Ralph Wiggum, three million bytes from Tolstoy, some excerpts from Machiavelli and Shakespeare, the entirety of the King James V Bible, and more. So that you know what to expect, open and skim each of those files, as with `gedit`. For instance, to open `austinpowers.txt`, open a terminal window and execute the below.

```
gedit ~cs50/pset5/texts/austinpowers.txt
```

Alternatively, launch `gedit`, select **File > Open…**, click **Computer** at left (under **Devices**), double-click **home** at right, click **cs50** at right, double-click **pset5** at right, double-click **texts** at right, then double-click **austinpowers.txt** at right. (If you get lost, simply start these steps over!)

Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if executed with, say,

```
./speller ~cs50/pset5/texts/austinpowers.txt
```

will eventually resemble the below. For now, try executing the staff's solution (using the default dictionary) with the below.

```
~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt
```

Below's some of the output you'll see. For amusement's sake, we've excerpted some of our favorite "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
Bigglesworth
[...]
Fembots
[...]
Virtucon
[...]
friggin'
[...]
shagged
[...]
trippy
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` represents the number of seconds that `speller` spends executing your implementation of `load`. `TIME IN check` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `TIME IN size` represents the number of seconds that `speller` spends executing your implementation of `size`. `TIME IN unload` represents the number of seconds that `speller` spends executing your implementation of `unload`. `TIME IN TOTAL` is the sum of those four measurements.

Note that these times may vary somewhat across executions of `speller`, depending on what else your appliance is doing, even if you don't change your code.

Incidentally, to be clear, by "misspelled" we simply mean that some word is not in the `dictionary` provided.

And now this:

http://www.youtube.com/watch?v=RIevazPIPzU

Spell Checking

Alright, the challenge now before you is to implement `load`, `check`, `size`, and `unload` as efficiently as possible, in such a way that `TIME IN load`, `TIME IN check`, `TIME IN size`, and `TIME IN unload` are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed `speller` different values for `dictionary` and for `text`. But therein lies the challenge, if not the fun, of this problem set. This problem set is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us.

- You may not alter `speller.c`.
- You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `check`, `size`, and `unload`), but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
- You may alter `Makefile`.
- You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
- Your implementation of `check` must be case-insensitive. In other words, if `foo` is in dictionary, then `check` should return true given any capitalization thereof; none of `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo`, and `FOO` should be considered misspelled.
- Capitalization aside, your implementation of `check` should only return `true` for words actually in `dictionary`. Beware hard-coding common words (e.g., `the`), lest we pass your implementation a `dictionary` without those same words. Moreover, the only possessives allowed are those actually in `dictionary`. In other words, even if `foo` is in `dictionary`, `check` should return `false` given `foo’s` if `foo’s` is not also in `dictionary`.
- You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.

- You may assume that any `dictionary` passed to your program will be structured exactly like ours, lexicographically sorted from top to bottom with one word per line, each of which ends with `\n`. You may also assume that `dictionary` will contain at least one word, that no word will be longer than `LENGTH` (a constant defined in `dictionary.h`) characters, that no word will appear more than once, and that each word will contain only lowercase alphabetical characters and possibly apostrophes.
- Your spell-checker may only take `text` and, optionally, `dictionary` as input. Although you might be inclined (particularly if among those more comfortable) to "pre-process" our default dictionary in order to derive an "ideal hash function" for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell-checker in order to gain an advantage.
- You may research hash functions in books or on the Web, so long as you cite the origin of any hash function you integrate into your own code.

Alright, ready to go?

load

Implement `load`!

Allow us to suggest that you whip up some dictionaries smaller than the 143,091-word default with which to test your code during development. And here’s Zamyla with some additional guidance:

http://www.youtube.com/watch?v=E_1D17P-bM0

check

Implement `check`!

Allow us to suggest that you whip up some small files to spell-check before trying out, oh, War and Peace. And here’s Zamyla again:

http://www.youtube.com/watch?v=r7CVY6O-XJw

size

Implement `size`!

If you planned ahead, this one is easy! Here's Zamyla!

http://www.youtube.com/watch?v=lO8k5CjLk4U

unload

Implement `unload`!

Be sure to free any memory that you allocated in `load`! Here's Zamyla with some final suggestions!

http://www.youtube.com/watch?v=VodrNEvdBBc
In fact, be sure that your spell-checker doesn't leak any memory at all. Recall that `valgrind` is your newest best friend. Know that `valgrind` watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want `valgrind` to analyze `speller` while you use a particular `dictionary` and/or text, as in the below.

```
valgrind --leak-check=full ./speller ~cs50/pset5/texts/austinpowers.txt
```

If you run `valgrind` without specifying a `text` for `speller`, your implementations of `load` and `unload` won't actually get called (and thus analyzed).

And don't forget about your other good buddy, `gdb`.

Checking Spell Checking

How to check whether your program is outting the right misspelled words? Well, you're welcome to consult the "answer keys" in `~cs50/pset5/keys`, as with the below.

```
gedit ~cs50/pset5/keys/austinpowers.txt
```

Alternatively, you could run your program on some text in one window, as with the below.

```
./speller ~cs50/pset5/texts/austinpowers.txt
```

And you can then run the staff's solution on the same text in another window, as with the below.

```
~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program’s output to a file (just like you may have done with `generate` in Problem Set 3), as with the below.

```
./speller ~cs50/pset5/texts/austinpowers.txt > student.txt
~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt > staff.txt
```

You can then compare both files side by side in the same window with a program like `diff`, as with the below.

```
diff -y student.txt staff.txt
```

Alternatively, to save time, you could just compare your program’s output (assuming you redirected it to, e.g., `student.txt`) against one of the answer keys without running the staff’s solution, as with the below.

```
diff -y student.txt ~cs50/pset5/keys/austinpowers.txt
```

If your program’s output matches the staff’s, `diff` will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you’ll see a `>` or `|` where they differ. For instance, if you see

```
MISSPELLED WORDS                                                MISSPELLED
 WORDS

FOTTAGE                                                         FOTTAGE
INT                                                             INT
                                                              > EVIL'S
s                                                               s
                                                              > EVIL'S
Farbissina                                                      Farbissina
```

that means your program (whose output is on the left) does not think that `EVIL’s` is misspelled, even though the staff’s output (on the right) does, as is implied by the absence of `EVIL’s` in the lefthand column and the presence of `EVIL’s` in the righthand column.

To test your code less manually (though still not exhaustively), you may also execute the below:

```
check50 2014.fall.pset5.speller dictionary.c dictionary.h Makefile
```

Note that `check50` does not check for memory leaks, so be sure to run `valgrind` as prescribed as well.

How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
~cs50/pset5/speller ~cs50/pset5/texts/austinpowers.txt
```

Big Board

Afraid the Big Board is only available on campus!

questions.txt

Congrats! At this point, your speller-checker is presumably complete (and fast!), so it's time for a debriefing. In `questions.txt`, answer each of the following questions in a short paragraph.

7. What data structure(s) did you use to implement your spell-checker? Be sure not to leave your answer at just "hash table," "trie," or the like. Expound on what's inside each of your "nodes."
8. How slow was your code the first time you got it working correctly?
9. What kinds of changes, if any, did you make to your code in order to improve its performance?
10. Do you feel that your code has any bottlenecks that you were not able to chip away at?

How to Submit

Step 1 of 2

When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset5` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset5`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset5.zip pset5
```

If you type `ls` thereafter, you should see that you have a new file called `pset5.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset5.zip`, then create a new ZIP file as before.) * Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.edx.org/submit[1], logging in if prompted. * Click **Submit** toward the window's top-left corner. * Under **Problem Set 5** on the screen that appears, click **Upload New Submission**. * On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear. * Navigate your way to `pset5.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset5.zip`, click it once to select it, then click **Open**. * Click **Start upload** to upload your ZIP file to CS50's servers. * On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.edx.org/submit[2] and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

Head to http://cs50.edx.org/2015/psets/5/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 5.

[1] http://cs50.edx.org/submit
[2] http://cs50.edx.org/submit