Problem Set 6: Mispellings

due by noon on Thu 10/27

Per the directions at this document's end, submitting this problem set involves submitting source code via `submit50` as well as filling out a Web-based form, which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

- Allow you to design and implement your own data structure.
- Optimize your code's (real-world) running time.
- Learn how to use version control.
- Challenge the BIG BOARD.

Recommended Reading.

- Sections 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.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the course's instructor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Admonish*, *Probation*, *Requirement to Withdraw*, or *Recommendation to Dismiss*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Grades.

Your work on this problem set will be evaluated along four axes primarily.

Scope. To what extent does your code implement the features required by our specification?
Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

All students, whether taking the course Pass/Fail or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a passing grade (*i.e.*, Pass or A to D–) unless granted an exception in writing by the course's instructor.

Getting Started.

☐ Launch VirtualBox (as by double-clicking its icon wherever it's installed), and then boot the appliance (as by single-clicking it in VirtualBox's lefthand menu, and then clicking **Start**).

Upon reaching John Harvard's desktop, open a terminal window (remember how?) and type the below, followed by Enter:

```
sudo yum -y update
```

Input **crimson** if prompted for John Harvard's password. For security, you won't see any characters as you type. Realize that updating the appliance in this manner requires Internet access. If on a slow connection (or computer), it might take a few minutes to update the appliance. Don't worry if the process seems to hang if it decides to update a "package" called `cs50-appliance`; that one can take several minutes.

If you see messages like **Couldn't resolve host** or **Cannot retrieve metalink for repository**, those simply mean that the appliance doesn't currently have Internet access. Sometimes that happens if you've just awakened your computer from sleep or perhaps changed from wireless to wired Internet or vice versa. If your own computer does have Internet access (which you can confirm by trying to visit some website in a browser on your own computer) but the appliance does not (which you can confirm by trying to visit the same with Firefox within the appliance), try restarting the appliance (as by clicking the green icon in its bottom-right corner, then clicking **Restart**).[1] If, upon restart, the appliance still doesn't have Internet access, head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` for help!

Once the appliance has been updated, you should see **Complete!** in your terminal window. If there was nothing to update, you'll see **No packages marked for Update** instead.

☐ Just to be sure that everything worked, go ahead and execute that very same command again in a terminal window (though not while its first invocation is still running):

```
sudo yum -y update
```

Again, input **crimson** if prompted for John Harvard's password. (If only a few minutes have passed since the last update, you might not even be prompted.) You should now see **No packages marked for Update**, which means that your appliance is now up-to-date! If you see some error instead, try once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

[1] Alternatively, you can try typing:
`sudo service network restart`
But if that doesn't work, best to restart the appliance.

☐ Now go ahead and open up a terminal window (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

```
cd
```

to ensure that you're in your home directory, and then execute

```
git clone http://cdn.cs50.net/2011/fall/psets/6/pset6.git
```

to download this problem set's distro into your appliance. You should see **`Cloning into pset6...`** and then your prompt again. If you instead see **`fatal`** followed by **`not found: did you run`**, odds are you made a typo. Best to try again!

Once successful, you should find that you have a brand-new `pset6` directory inside of your home directory. You can confirm as much with:

```
ls
```

Navigate your way to that directory by executing the command below.

```
cd ~/pset6
```

If you list the contents of your current working directory (remember how?), you should see the below. If you don't, don't hesitate to ask the staff for assistance!

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

Alotta Mispellings.

☐ 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 seconds—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.

☐ 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/pset6/dictionaries/large` by default. In other words, running

```
./speller text
```

will be equivalent to running

```
./speller ~cs50/pset6/dictionaries/large text
```

where `text` is the file you wish to spell-check.[2] Suffice it to say, the former is easier to type!

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/pset6/dictionaries/small` is one such dictionary. To use it, execute

```
./speller ~cs50/pset6/dictionaries/small text
```

where `text` is the file you wish to spell-check.[3] Don't move on until you're sure you understand how `speller` itself works!

[2] 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**.
[3] *Ibid*.

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

☐ Okay, technically that last problem induced an infinite loop. But we'll assume you broke out of it. In `questions.txt`, 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`?

☐ Now take a look at `Makefile`. Notice that we've employed some new tricks. Rather than hard-code specifics in targets, we've instead defined variables (not in the C sense but in a `Makefile` sense). The line below defines a variable called `CC` that specifies that make should use `gcc` for compiling.

```
CC = gcc
```

The line below defines a variable called `CFLAGS` that specifies, in turn, that `gcc` should use some familiar flags.

```
CFLAGS = -ggdb -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 in the semester.) 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 default target using these variables.

```
$(EXE): $(OBJS)
    $(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 a 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`.

Even though this `Makefile` is fancier than past ones, compiling your code remains as easy as executing the below.

```
make
```

But now you know how to make (pun intended) a more sophisticated `Makefile`!

☐ On to the most fun of these files! Notice that in `~cs50/pset6/texts/`, we have provided you with a whole bunch of texts with which you'll be able to test your implementation of `speller`. Among those files are 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/pset6/texts/austinpowers.txt
```

Alternatively, double-click **File System** on John Harvard's desktop, then double-click **home**, then double-click **cs50**, then double-click **pset6**, then double-click **texts**, then double-click **austinpowers.txt**.

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

```
./speller ~cs50/pset6/texts/austinpowers.txt
```

will eventually resemble the below.[4] For now, try executing the staff's solution (using the default dictionary) with the below.

```
~cs50/pset6/speller ~cs50/pset6/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.

[4] *Ibid*.

Incidentally, to be clear, by "misspelled" we mean that some word is not in the `dictionary` provided. "Fembots" might very well be in some other (swinging) dictionary.

☐ Alright, the challenge ahead of 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.

i. You may not alter `speller.c`.
ii. 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`.
iii. You may alter `dictionary.h`, but you may not alter the declarations of `load`, `check`, `size`, or `unload`.
iv. You may alter `Makefile`.
v. You may add functions to `dictionary.c` or to files of your own creation so long as all of your code compiles via `make`.
vi. 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.
vii. 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`.
viii. You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.
ix. 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 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.
x. 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.
xi. 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?

☐ 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.

☐ Implement `check`!

Allow us to suggest that you whip up some small files to spell-check before trying out, oh, War and Peace.

☐ Implement `size`!

If you planned ahead, this one is easy!

☐ Implement `unload`!

Be sure to free any memory that you allocated in `load`!

☐ 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 -v --leak-check=full ./speller ~cs50/pset6/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).

☐ Don't forget about your other good buddy, `gdb`.

☐ And `help.cs50.net`.

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

```
~cs50/pset6/speller ~cs50/pset6/texts/austinpowers.txt
```

But also feel free to put your code to the test against your own classmates'! Execute the command below to challenge the BIG BOARD starting Mon 10/24.

```
check50 ~/pset6
```

We'll benchmark your spell-checker with a variety of inputs. Assuming your output's correct, you can then surf on over to the course's home page to see how your `speller` stacks up against others'! Feel free to challenge the BIG BOARD as often as you'd like; it will display your most recent results.

We shall honor those atop the BIG BOARD.

By the way, you might want to turn off GCC's `-ggdb` flag when challenging the BIG BOARD. And you might want to read up on GCC's `-O` flags! (Remember how?)

Those more comfortable might also find such tools as `gprof` and `gcov` of interest.

☐ 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 over the course of the week 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.

In order to submit this problem set, you must first execute a command in the appliance and then submit a (brief) form online.

☐ Recall that you obtained a CS50 Cloud account (*i.e.*, username and password) for Problem Set 1. If you don't remember your username and/or password, head to `https://cloud.cs50.net/` look up the former and/or change the latter. You'll be prompted to log in with your HUID (or XID) and PIN.

☐ Just in case we updated `submit50` since you started this problem set, open a terminal window and execute the below, inputting **crimson** if prompted for John Harvard's password.

```
sudo yum -y update
```

If there was something to update, you should see **Complete!** after a few seconds or minutes. If there was nothing to update, you should instead see **No packages marked for Update**. If you see any errors, try the command once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ To actually submit, first open a terminal window and execute:[5]

```
cd ~/pset6
```

[5] Unless you decided to use `dropbox.com` and stored your files in, say, `~/Dropbox/pset6`.

Then execute:

```
ls
```

At a minimum, you should see `dictionary.c`, `dictionary.h`, `Makefile`, `questions.txt`, and `speller.c`. If not, odds are you skipped some more steps earlier! If everything is as it should be, you are ready to submit your source code to us. Execute:[6]

```
submit50 ~/pset6
```

When prompted for **Course**, input **cs50**; when prompted for **Repository**, input **pset6**. When prompted for a username and password, input your CS50 Cloud username and password. For security, you won't see your password as you type it. That command will essentially upload your entire `~/pset6` directory to CS50's repository, where your TF will be able to access it. The command will inform you whether your submission was successful or not. If provided with the URL of a PDF of your code (which further confirms its submission), right-click (or ctrl-click) the link, then choose **Open Link** from the menu that appears to open the PDF in Document Viewer.

You may re-submit as many times as you'd like; we'll grade your most recent submission. But take care not to submit after the problem set's deadline, lest you spend a late day unnecessarily or risk rejection entirely.

If you run into any trouble at all, let us know via `help.cs50.net` and we'll try to assist! Just take care to seek help well before the problem set's deadline, as we can't always reply within minutes!

☐ Anytime after lecture on Mon 10/24 but before this problem set's deadline, head to the URL below where a short form awaits:

```
https://www.cs50.net/psets/6/
```

Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 6.

[6] *Ibid.*