NOTICE TO PODCAST SUBSCRIBERS

The source code for CS 50's library as well as for Fall 2007's problem sets in general can be found at `http://cs50.tv/`.

Problem Set 5: Mispellings[1]

out of 98 points

due, in part, by 7:00 P.M. on Tuesday, 13 November 2007, per #9
due, in full, by 7:00 P.M. on Friday, 16 November 2007

Goals.

The goals of this problem set are to:

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

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

- Section 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*.

[1] Yes, we know. It's meant to be ironic. Or something.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). 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, magazine, 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.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. 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.

All forms of cheating will be dealt with harshly.

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 staff.

Grading Metrics.

Each question is worth the number of points specified parenthetically in line with it.

Your responses to questions requiring exposition will be graded on the basis of their clarity and correctness. Your responses to questions requiring code will be graded along the following axes.

> **Correctness.** To what extent does your code adhere to the problem's specifications?
> **Design.** To what extent is your code written clearly, efficiently, elegantly, and/or logically?
> **Style.** To what extent is your code commented and indented, your variables aptly named, *etc.*?

Rest assured that grades will be normalized across sections at term's end.

Getting Started.

0. SSH to `nice.fas.harvard.edu` and recursively copy `~cs50/pub/ps/distributions/ps5` into your own `~/cs50` directory. (Remember how?)

   Navigate your way to `~/cs50/ps5/`. If you list the contents of your current working directory, you should see the below. If you don't, don't hesitate to ask the staff for assistance.

```
design.txt     dictionary.h  questions.txt    speller.c
dictionary.c   Makefile      reflections.txt  texts/
```

1. (12 points.) Surf on over to the URL below.

   `http://www.fas.harvard.edu/~cs50/surveys/ps5/`

   Please take some time to provide candid answers to the survey's questions. Although you will be prompted to authenticate with your HUID and PIN, your answers, immediately upon submission, will be anonymized. We will know that you took the survey, but we will not know which answers are yours.

Alot of Mispellings.

2. 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, real 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 143,090-word dictionary from disk into memory. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both we leave to you!

   Before we walk you through `speller.c`, go ahead and open up `dictionary.h` with Nano. Declared in that file are three functions; take note of what each should do. Now open up `dictionary.c`. Notice that we've implemented those three 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.

3. Open up `speller.c` with Nano 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`, and `size`. 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

   ```
   speller [dict] file
   ```

   where `dict` is assumed to be a file containing a list of lowercase words, one per line, and `file` is a file to be spell-checked. As the brackets suggest, provision of `dict` is optional; if this argument is omitted, `speller` will use `/home/c/s/cs50/pub/share/dict/words` by default for its dictionary. Within that file are those 143,090 words that you must ultimately load. In fact, take a peek at that file with Nano (or `more` or `less`) to get a sense of its structure and size. Notice that every word in that file indeed appears in lowercase (even, for simplicity, proper nouns and acronyms), one per line. During development, you may find it helpful to provide `speller` with a `dict` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory.

   Don't move on until you're sure you understand how `speller` itself works!

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

5. (8 points.) 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.

   i. According to its man page, what does `getrusage` do?
   ii. How many members are in a variable of type `struct rusage`?
   iii. Why do you think we pass `before` and `after` by reference to `calculate`, even though we're not changing their contents?
   iv. 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.
   v. 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. 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
   ```

   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 `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, by way of these variables, a default target, the output of which will be `speller`.

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

   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'll need to if you create any `.c` or `.h` files of your own.

7. On to the most fun of these files! Notice that in `ps5/texts/`, we have provided you with a number of texts with which you'll be able to test your `speller`. Among those files are the script from *Austin Powers: International Man of Mystery* (yeah, baby, yeah!), a sound bite from Ralph Wiggum, three million bytes from Tolstoy, some excerpts from Machiavelli and Shakespeare, and the entirety of the King James V Bible. So that you know what to expect, take a peek at those files using `cat`, `less`, or `more`. You can also use Nano, but be sure not to save any changes accidentally.

   Now, as you should know from having read over `speller.c` carefully, the output of `speller`, if run on, say, `austinpowers.txt`, should resemble the below. 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 FILE:
STARTUP TIME:
LOOKUP TIME:
SIZING TIME:
TOTAL TIME:
```

   `STARTUP TIME` represents the number of seconds that `speller` spends executing your implementation of `load`. `LOOKUP TIME` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `SIZING TIME` represents the number of seconds that `speller` spends executing your implementation of `size`. `TOTAL TIME` is the sum of those three measurements.

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

8. Alright, the challenge ahead of you is to implement `load`, `check`, and `size` as efficiently as possible, in such a way that `STARTUP TIME`, `LOOKUP TIME`, and `SIZING TIME` 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 `dict` and for `file`. 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 not alter the declaration of `load`, `check`, or `size` in `dictionary.{c,h}`.
    iii. You must alter the implementations of `load`, `check`, and `size` in `dictionary.c`.
    iv. You may alter `dictionary.h` and `Makefile`.
    v. You may implement functions other than `load`, `check`, and `size` (in `dictionary.{c,h}` or new files) in order to facilitate your implementation of the same.
    vi. Your implementation of `check` must be case-insensitive. In other words, if `foo` is in `dict`, 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 `dict`. Beware hard-coding common words (*e.g.*, `the`), lest we pass your implementation a `dict` without those same words. Moreover, the only possessives allowed are those actually in `dict`. In other words, even if `foo` is in `dict`, `check` should return `FALSE` given `foo's` if `foo's` is not also in `dict`.
    viii. You may assume that `check` will only be passed strings with alphabetical characters and/or apostrophes.
    ix. You may assume that `dict` will only contain lowercase alphabetical characters, apostrophes, and/or newlines and that no word in `dict` will be longer than `LENGTH` (a constant defined in `speller.c`).

9. (12 points.) With this problem set's design so important, allow us to help you help yourself by asking that you write your first "design document" for this course. In `ps5/design.txt`, answer each of the questions below.

    i. Describe in detail in English the data structure(s) with which you plan to implement this spell-checker. Also define the data structure(s) in C (as with `struct`).
    ii. Tell us, in pseudocode, step by step, how you intend to implement `load`.
    iii. Tell us, in pseudocode, step by step, how you intend to implement `check`.
    iv. Tell us, in pseudocode, step by step, how you intend to implement `size`.

    Email `design.txt` as an attachment to your teaching fellow **by 7:00 P.M. on Tuesday, 13 November 2007**.[2] You need not wait for a reply before you proceed with implementation. Consider this requirement an opportunity for counsel. Include any questions you have in the body of your email. It's absolutely fine if you alter your design as you proceed with implementation. In fact, it is quite likely you will.

[2] If you plan to spend *n* late days on this problem set, you must email your design document to your teaching fellow no more than 24*n* hours after this deadline in order to receive credit.

10. (30 points.) Implement `load`!

    Allow us to suggest that you whip up some dictionaries smaller than the 143,090-word default with which to test your code during development.

11. (30 points.) Implement `check`!

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

12. (3 points.) Implement `size`!

    If you planned ahead, this one is easy!

13. How to assess just how fast (and correct) your code is? Well, feel free to play with the staff's solution in `~cs50/pub/ps/solutions/ps5/`. But also feel free to put your code to the test against your own classmates'! Execute the command below to challenge THE BIG BOARD.

    ```
    bigboard
    ```

    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.[3]

    We shall honor those atop THE BIG BOARD.

    Realize that, for convenience, THE BIG BOARD includes links to lists of words considered misspellings (with respect to our specifications and that 143,090-word dictionary) for each of the texts in `ps5/texts/`.

    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.

14. (3 points.) And now let's have you reflect in `ps5/reflections.txt`. Describe in detail the data structure(s) with which you actually implemented this spell-checker. In what way(s) did your actual implementation ultimately differ from the design you put forth in `design.txt`? Why did you diverge from your original design?

[3] Realize, incidentally, that your spell-checker's performance might very well vary based on what others are doing on `nice.fas.harvard.edu` at the moment you challenge. That reality, however, is part of the challenge! If you're determined to fight load or find better hardware, you're welcome to SSH to specific machines within the `nice.fas.harvard.edu` cluster.

Submitting Your Work.

15. Ensure that your work is in `~/cs50/ps5/`. Submit your work by executing the command below.

    ```
    cs50submit ps5
    ```

    Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.

    Per #1, don't forget to complete the survey at the URL below!

    `http://www.fas.harvard.edu/~cs50/surveys/ps5/`