Problem Set 6: Mispellings

due by 7:00pm on Fri 10/29

Per the directions at this document's end, submitting this problem set involves submitting source code on `cloud.cs50.net` as well as filling out a Web-based form (the latter of which will be available after lecture on Wed 10/27), 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. 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 *Warn*, *Admonish*, or *Disciplinary Probation*, 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 three primary axes.

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)?

Getting Started.

☐ Hi! SSH to `cloud.cs50.net` and execute the commands below.

```
mkdir ~/pset6/
cp -a ~cs50/pub/src/psets/pset6/* ~/pset6/
```

Now pull up the man page for `cp`. Notice that this `-a` switch is defined as follows.

```
-a, --archive
       same as -dpR
```

Those lines mean that running `cp` with `-a` (or `--archive`) is the same as running `cp` with `-dpR` (or, equivalently, `-d -p -R`). The last of those switches (`-R`) should be familiar, as it's equivalent to `-r`, which you've used in past problem sets to copy directories recursively. The second of those switches (`-p`) tells `cp` to preserve files' modes (*i.e.*, permissions) and timestamps. The first of the switches, finally, means that `cp` should not de-reference (*i.e.*, follow) "symbolic links." A symbolic (also known as a symlink or soft link) is a special kind of file that pretty much just contains the path to another file. In fact, let's take a look at what you just copied. Navigate your way to `~/pset6/` and type the (familiar) command below.

```
ls
```

You should see the below.

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

Notice that `texts` has a trailing `@`. Now execute the (also familar) command below.

```
ls -l
```

You should see output like the below (if you happen to be John Harvard).

```
-rw-r--r-- 1 jharvard students  538 Oct 22 18:59 Makefile
-rw-r--r-- 1 jharvard students  971 Oct 22 18:59 dictionary.c
-rw-r--r-- 1 jharvard students  990 Oct 22 18:59 dictionary.h
-rw-r--r-- 1 jharvard students    0 Oct 22 18:59 questions.txt
-r--r--r-- 1 jharvard students 5205 Oct 22 18:59 speller.c
lrwxrwxrwx 1 jharvard students   32 Oct 22 18:59 texts -> /home/cs50/pub/share/pset6/texts/
```

As suggested by the output above, `texts` is indeed a symlink that happens to lead to `/home/cs50/pub/share/pset6/texts/`.

Why all this talk about symlinks? Well, rather than waste space by giving each of you your own copy of `texts` (inside of which is a whole bunch of read-only files), we decided to put one copy in the cloud but make it easy for you to get at. In other words, because you have a symlink to `/home/cs50/pub/share/pset6/texts/` in your `~/pset6/` directory, you'll be able to access

files therein by way of a much shorter relative path. In fact, let's bring this point home. Assuming you're still in `~/pset6/`, go ahead and execute the below.[1]

```
ls texts/
```

You should see a whole bunch of (fun) files, as though `texts` were, in fact, a normal directory inside your `~/pset6/` directory. Even though it's not!

So, why was `-a` (and, in turn, `-d`) useful with `cp`? Well, that flag ensured that the symlink to `/home/cs50/pub/share/pset6/texts/` was not "followed" but instead copied as a symlink, else you'd end up with your own copy of that directory (since that's where you end up by following the symlink).

Want to procrastinate (and practice "following" a symlink)? Try out the command below from within `~/pset6/`.[2]

```
more texts/austinpowers.txt
```

Yeah, baby, yeah!

☐ Odds are you've lost some important file at least once in your life, whether because you accidentally deleted it or because your hard drive failed. So you may already be in the habit of making backups of your code. But you've probably been doing it the old-fashioned way, by making copy after copy of your code, giving each file its own name? Turns out there's a better way, generally called "version control," that doesn't involve littering your current working directory. With version control, anytime you want to create a backup of some file, you run a command to "commit" or "check in" that file to a "repository," which is a special directory on some server or in your current working directory, perhaps hidden, in which all of your backups reside. You can then restore any one of those backups at anytime with a simple command. And you can even annotate each of your backups so that you actually remember how it differs from others.

There are a ton of version control systems out there, some of which are considered "distributed" in the sense that they don't require a central server. Some of the most popular version control systems at the moment are Bazaar, Git, Mercurial, and Subversion; older favorites include CVS and RCS.

You're welcome to use any system you'd like, but allow us to recommend that you use Mercurial (aka `hg`) for this problem set and beyond, as it's both simple and powerful.[3] Let's take it out for a spin.

[1] Be sure to include that trailing slash (else you'll just be reminded that `texts` is a symlink)!
[2] Hit `q` to quit out of `more`!
[3] Recall that `hg` is the symbol for mercury. Clever, eh?

If not already there, navigate your way to `~/pset6/`, as with the command below.

```
cd ~/pset6/
```

Now execute this command to initialize (*i.e.*, create) a repository for this problem set's code:

```
hg init
```

Let's see what that command did. Go ahead and execute `ls`, just as you did before. You should see the below.

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

Hm, doesn't look any different. Go ahead and execute

```
ls -a
```

instead, and you should instead see the below.

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

Oh interesting. Recall that `-a` signifies "all," and so you now see absolutely everything in this directory, including files and directories whose names start with a dot (`.`), which are normally hidden by default. Recall that `.` denotes your current directory and `..` its parent. But `.hg` is new. That's a "hidden" directory that `hg` just created for you; it's the so-called "repository" in which all of your revisions will go. It's hidden, though, because `hg` doesn't want you poking around there. Know that it exists, but don't go there, as they say.

How to start making backups then? Go ahead and execute the below.

```
hg add
```

You should see the output below:

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

Suffice it to say that each of the files in your current working directory have been added to the repository, which means that Mercurial will henceforth keep an eye on them for you. But that doesn't mean they've actually been backed up, just that Mercurial now knows which files to back up. Let's actually back up these files. Go ahead and execute the command below.

```
hg commit
```

You should find that `nano` is now running, with your cursor positioned above the following line, where `username` is your own username.

```
HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: username
HG: branch 'default'
HG: added Makefile
HG: added dictionary.c
HG: added dictionary.h
HG: added questions.txt
HG: added speller.c
HG: added texts
```

Mercurial has launched `nano` because it wants you to associate a "commit message" (*i.e.*, notes to yourserlf) with this backup, so that you remember (at, oh, 2am) what this version of your code does. Since this is the first time you're committing these files, write yourself an obvious note like

```
This is Problem Set 6's distribution code.
```

above all those `HG` lines, then save and quit `nano` as usual: hit ctrl-x, then Enter when prompted to save the modified buffer, then Enter again when prompted for a file name to write. Take care not to change the (weird) temporary filename you see; just hit Enter. And take care not to leave the commit message itself blank, else Mercurial will abort the commit altogether. Once you've committed the code, you should find yourself back at a prompt. Confirm that the commit was successful by executing the command below.

```
hg log
```

You should see output resembling the below.

```
changeset:   0:13d2516423d8
tag:         tip
user:        username
date:        Fri Oct 22 19:00:01 2010 -0400
summary:     This is Problem Set 6's distribution code.
```

Looks like Mercurial indeed logged a bunch of details about the commit.

Let's next do something daring. Go ahead and execute

```
rm Makefile
```

followed by `y` when prompted to remove that file. If you execute `ls`, you'll see that the file is indeed gone.

OMG why did you listen to us? Not to worry, you can restore your last checked-in version of Makefile by executing the below:

```
hg revert Makefile
```

Phew, if you execute `ls` again, you should see that it's back.[4] Now let's try something more interesting. Go ahead and open `questions.txt` with `nano`, and type in some gibberish like the below.

```
asdfghjkl
```

Then save and quit `nano` with ctrl-x as usual. Suppose that you're pretty content with that answer, and so you want to commit this change to the repository. Go ahead and execute

```
hg commit
```

as before. Input a message to yourself like

```
Finished questions.txt.
```

when prompted, then save and quit `nano` with ctrl-x as before. Then execute

```
hg log
```

again. You should see output resembling the below.

```
changeset:   1:9c2c0bba179d
tag:         tip
user:        username
date:        Fri Oct 22 19:05:01 2010 -0400
summary:     Finished questions.txt.

changeset:   0:13d2516423d8
user:        username
date:        Fri Oct 22 19:00:01 2010 -0400
summary:     This is Problem Set 6's distribution code.
```

Notice that the log is sorted, from top to bottom, in reverse chronological order. And notice that the earliest commit (*i.e.*, changeset) is identified labeled with `0:13d2516423d8`. That means you can restore that particular backup via its "revision number" (`0`) or its "nodeid" (`13d2516423d8`). Odds are you'll find the former simpler, so go ahead and execute the below:

```
hg update 0
```

[4] If not, you must have skipped a step somewhere! You can grab a new copy of `Makefile` by executing this command:
`cp ~cs50/pub/src/psets/pset6/Makefile ~/pset6/`

If you open `questions.txt` with `nano`, you should find that it's back to its original, empty state. In other words, by executing

```
hg update 0
```

you just reverted all of the files in your current working directory (one of which is, of course, `questions.txt`) to whatever they looked like when you first committed them.

You can think of Mercurial, then, as a time machine of sorts. Anytime you want to back up your code, execute:

```
hg commit
```

Anytime you want to roll back in time, first check Mercurial's log with

```
hg log
```

to figure out the revision number (or nodeid) of the backup to which you'd like to revert, then execute

```
hg update #
```

where `#` is the revision number (or nodeid) you want. That will revert every file in your current working directory to whatever it looked like when you performed that commit. If you only want to undo changes you've made to a specific file since your most recent commit, you can instead execute

```
hg revert filename
```

to go back in time for that file, `filename`, only.

We've actually just scratched the surface of Mercurial here, but these fundamentals alone should be enough to save you some time (if not tears!) over the next several weeks. If you'd like to learn some additional tricks on your own, you might like to peruse the below.

```
http://mercurial.selenic.com/wiki/BeginnersGuides
```

☐ Let's now add one additional tool to your toolbox.

Although `nano` is one of the simplest editors that can be found on a Linux system, it's definitely lacking in features as a result. And so it's time to transition to something more user-friendly. In fact, now that you know about SFTP, you can actually start using a "client-side" text editor on your own laptop that saves your files not on your hard drive but on `cloud.cs50.net`, where you can then compile and execute it as usual.

For this problem set, you're welcome to code using any text editor you'd like (including `nano` if you've fallen in love) or even an integrated development environment (IDE), but allow us to

suggest that Mac users use TextWrangler and PC users use Notepad++.[5] Head to the appropriate URL below to learn how to use one or the other.

Mac users: `http://wiki.cs50.net/TextWrangler`
PC users: `http://wiki.cs50.net/Notepad%2B%2B`

Note that both TextWrangler and Notepad++ come with support for SFTP built-in, so you don't need to use a separate SFTP client (like you did for Problem Set 5). TextWrangler and Notepad++ will automatically save any changes you make to this problem set's files to `cloud.cs50.net` for you, so long as you follow the directions above.

To be clear, although you can now write your code with TextWrangler or Notepad++, you'll still want to SSH to `cloud.cs50.net` in order to run `gcc`, `gdb`, `make`, `submit`, `valgrind`, and any other commands.

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 143,091-word dictionary 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 TextWrangler or Notepad++. 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 TextWrangler or Notepad++ 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

[5] If you're a Linux user, odds are you already have a text editor of choice, but feel free to ask for suggestions via `help.cs50.net`.

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/cs50/pub/share/pset6/dict/words` by default for its dictionary. Within that file are those 143,091 words that you must ultimately load into memory. In fact, take a peek at that file to get a sense of its structure and size. 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 `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!

☐ 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?

☐ 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 `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)
```

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

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

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.
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 `~/pset6/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*, 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.

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:
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.

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 (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 `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 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 `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 any `dict` 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 `file` and, optionally, `dict` 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. Remember 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 `dict` and/or `file`, as in the below.

```
valgrind -v --leak-check=full ./speller texts/austinpowers.txt
```

If you run `valgrind` without specifying a `file` 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, feel free to play with the staff's solution, as in the below.

```
~cs50/pub/solutions/pset6/speller 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.

```
~cs50/pub/tests/pset6/challenge
```

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.[6]

We shall honor those atop THE BIG BOARD.

[6] Realize, incidentally, that your spell-checker's performance might very well vary based on what others are doing on `cloud.cs50.net` at the moment you challenge. That reality, however, is part of the challenge!

Realize that, for convenience, THE BIG BOARD includes links to lists of words considered misspellings (with respect to our specifications and that 143,091-word dictionary) for each of the texts in `~/pset6/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.

☐ 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.

6. Why do you think we declared so many parameters as being `const` in `dictionary.c` and `dictionary.h`?
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 (er, Thursday night) 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 on `cloud.cs50.net` and then submit a (brief) form online; the latter will be posted after lecture on Wed 10/27.

☐ SSH to `cloud.cs50.net`, if not already there, and then submit your code by executing:

```
~cs50/pub/bin/submit pset6
```

You'll know that the command worked if you are informed that your "work HAS been submitted." If you instead encounter an error that doesn't appear to be a mistake on your part, do try running the command one or more additional times. 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.

☐ Anytime after lecture on Wed 10/27 but before this problem set's deadline, head to the URL below where a short form awaits:

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

If not already logged in, you'll be prompted to log into the course's website.Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 6.