# Understanding

* 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*. Indeed, when describing the running time of an algorithm, we typically focus on the dominant (i.e., most impactful) term (i.e., *n* in this case, since *n* could be much larger than 2). In the real world, though, the fact of the matter is that 2*n* feels twice as slow as *n*.

# Dictionary.h

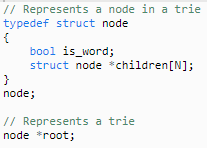
* Open up dictionary.h, and you’ll see some new syntax, including a few lines that mention DICTIONARY\_H



* + those lines just ensure that, even though dictionary.c and speller.c (which you’ll see in a moment) #include this file, clang will only compile it once.
  + #ifndef checks whether the given token, in this case DICTIONARY\_H, has been #defined earlier in the file or in an included file; if not, it includes the code between it and the closing #endif statement.
  + So, this way if we have defined DICTIONARY\_H before ifndef will prevent us from defining it again
* Next notice how we #include a file called **stdbool.h**. That’s the file in which bool itself is defined. You’ve not needed it before, since the CS50 Library used to #include that for you.
* Also notice our use of **#define**, a "preprocessor directive" that defines a "constant" called LENGTH that has a value of 45. It’s a constant in the sense that you can’t (accidentally) change it in your own code. In fact, clang will replace any mentions of LENGTH in your own code with, literally, 45. In other words, it’s not a variable, just a find-and-replace trick.
* Finally, notice the prototypes for four functions: check, load, size, and unload.
* **Const** just says that those strings, when passed in as arguments, must remain constant; you won’t be able to change them, accidentally or otherwise!

# Dictionary.c

* Notice how, atop the file, we’ve defined a struct called node that represents a node in a trie. And we’ve declared a global array, root, that represents the root (i.e., topmost node) of a trie.



* A bit below those lines, we implemented part of a function called load that will soon (thanks to you!) load a dictionary of words into that trie.
* We’ve written some code that initializes the trie with just one node at first for its root, each of whose children is initialized to NULL.
* And we’ve written some code that opens dictionary, which is the file name of a dictionary to load.
* And we’ve also written some code that iterates over that dictionary and reads the words therein, one at a time, into a buffer (i.e., string) called word. But we stop short of inserting those words into the trie. Thereafter, we do close the file, though, and then return true to indicate (we hope!) success.
* As for check, size, and unload, well, we’ve only just barely implemented those, enough for the file to compile.

# Speller.c

* Notice how, by way of a function called 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 dictionaries/large, a file in pset4, by default. In other words, running
  + ./speller text
* will be equivalent to running
  + ./speller 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.
  + 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 dictionaries/small is one such dictionary.

**Alternative if statement**

val = (expr) ? (vTrue) : vFalse ;

# text/

* 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
  + *La La Land*
  + the text of the Affordable Care Act
  + three million bytes from Tolstoy
  + some excerpts from *The Federalist Papers* and Shakespeare
  + the entirety of the King James V Bible and the Koran, and more.
* So that you know what to expect, open and skim each of those files, all of which are in a directory called texts within your pset4 directory.
* Now, as you should know from having read over speller.c carefully, the output of speller, if executed with, say,

./speller texts/lalaland.txt

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

~cs50/2019/x/pset4/speller dictionaries/large texts/lalaland.txt

* 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.
* **Note that these times may vary somewhat across executions of**speller**, depending on what else CS50 IDE 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.

# Makefile

* Makefile, a configuration file that tells make exactly what to do. Open up Makefile, and you should see three lines (not four), telling the computer that every time you type make or make speller the following happens:
  + tells make how to compile speller.c into machine code (i.e., speller.o).
  + tells make how to compile dictionary.c into machine code (i.e., dictionary.o).
  + tells make to link speller.o and dictionary.o in a file called speller.
* Note that you can copy all 3 lines into the terminal and they will run correctly