Problem Set 3: Scramble

due by noon on Thu 10/4

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.

Goals.

- Learn to read and build upon someone else's code.
- Learn how to encapsulate data.
- (Play.)

Recommended Reading.

- Chapters 13, 15, and 18 of *Programming in C*.

diff pset1.pdf hacker2.pdf.

- Hacker Edition requires a sort in $O(n \log n)$.
- Hacker Edition expects `INSPIRATION` instead of `SCRAMBLE`.
- Hacker Edition assigns different values to letters.

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 posted online) 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 course's instructor or preceptor.

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 or preceptor.

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.

Fine Print.

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 or preceptor. No more than one late day may be spent on this, or any other, problem set.

A Section of Questions.

You're welcome to dive into these questions on your own, but know that they'll also be explored in section!

☐ Head to

`https://www.cs50.net/shorts/`

and watch the shorts on GDB and binary search plus two or more of bubble sort, insertion sort, merge sort, and selection sort. (Phew, so many shorts! And so many sorts! Ha.) Be sure you're reasonably comfortable answering the below (even if you didn't watch all of the shorts) when it comes time to submit this problem set's form!

- ☐ GDB lets you "debug" a program, but, more specifically, what does it let you do?
- ☐ Why does binary search require that an array be sorted?
- ☐ Why is bubble sort in $O(n^2)$?
- ☐ Why is insertion sort in $\Omega(n)$?
- ☐ What's the worst-case running time of merge sort?
- ☐ In no more than 3 sentences, how does selection sort work?

☐ Using the CS50 Appliance, CS50 Run (`cs50.net/run`), or CS50 Spaces (`cs50.net/spaces`), type out the program below, then complete its implementation, per its `TODO`. Consider this problem an opportunity to practice; you won't be asked to submit this program.

```
#include <cs50.h>
#include <stdio.h>

#define SIZE 8

bool search(int needle, int haystack[], int size)
{
    // TODO: return true iff needle is in haystack, using binary search
}

int main(void)
{
    int numbers[SIZE] = { 4, 8, 15, 16, 23, 42, 50, 108 };
    printf("> ");
    int n = GetInt();
    if (search(n, numbers, SIZE))
        printf("YES\n");
    return 0;
}
```

☐ Using the CS50 Appliance, CS50 Run (`cs50.net/run`), or CS50 Spaces (`cs50.net/spaces`), type out the program below, then complete its implementation, per its `TODO`. Consider this problem an opportunity to practice; you won't be asked to submit this program.

```
#include <stdio.h>

#define SIZE 8

void sort(int array[], int size)
{
    // TODO: sort array using any algorithm in O(n log n)
}

int main(void)
{
    int numbers[SIZE] = { 4, 15, 16, 50, 8, 23, 42, 108 };
    for (int i = 0; i < SIZE; i++)
        printf("%d ", numbers[i]);
    printf("\n");
    sort(numbers, SIZE);
    for (int i = 0; i < SIZE; i++)
        printf("%d ", numbers[i]);
    printf("\n");
    return 0;
}
```

Getting Started.

☐ Welcome back!

Open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

```
sudo yum -y update
```

at your prompt to ensure that your appliance is up-to-date. In the future, you can instead execute

```
update50
```

instead.[1]

[1] We decided to simplify the process by writing our own update program for you!

☐ Recall that, for Problem Sets 1 and 2, you started writing programs from scratch, creating your own `pset1` and `pset2` directories with `mkdir`. For Problem Set 3, you'll instead download "distribution code" (otherwise known as a "distro"), written by us, and add your own lines of code to it. You'll first need to read and understand our code, though, so this problem set is as much about learning to read someone else's code as it is about writing your own!

Okay, go ahead and execute

```
mkdir ~/Dropbox/hacker3
```

in order to make a directory called `hacker3` in your `Dropbox` directory. Then execute

```
cd ~/Dropbox/hacker3
```

to move yourself into that directory. Your prompt should now resemble the below.

```
jharvard@appliance (~/Dropbox/hacker3):
```

If not, retrace your steps and see if you can determine where you went wrong!

Next execute

```
wget http://cdn.cs50.net/2012/fall/psets/3/hacker3/scramble.c
```

in order to download this problem set's distribution code. If you execute

```
ls
```

you should see that you indeed now have a file called `scramble.c` in your `hacker3` directory. (If not, be sure you didn't make a typo in that long URL!)

Lastly, execute

```
wget http://cdn.cs50.net/2012/fall/psets/3/hacker3/words
```

in order to download a dictionary with 172,806 English words. Confirm that it downloaded successfully by executing

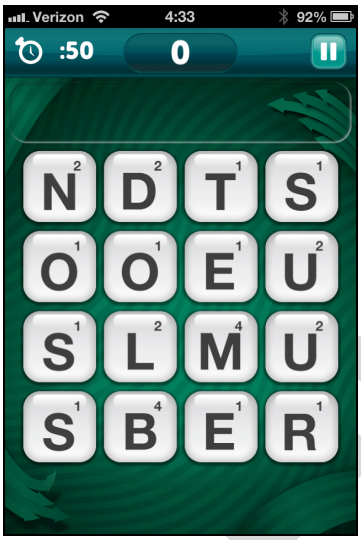
```
ls
```

one more time.

All of the work that you do for this problem set must ultimately reside in your `hacker3` directory for submission.

Scramble.

☐ And now it's time to play. Scramble is a game (currently popular on smart phones) that challenges you to find as many words as possible in a 4×4 grid of letters before a timer expires. Each pair of letters in a word can be adjacent horizontally, vertically, or diagonally. Below, for instance, is what the game looks like on an iPhone (at 4:33am). Present are words like LOTS, NO, NOD, and SUM, along with (believe it or not) 308 other words.



☐ Okay, open up a terminal window, if not open already, and execute

```
~cs50/hacker3/scramble
```

in order to try out the staff's implementation of `scramble`. You should see a 4×4 grid filled with letters. As soon as you spot a word, type it and hit Enter. If it's indeed a word in the grid (and in a dictionary of English words), your score will increase 1 point for each letter in the word. (Good job!) By default, you'll have 30 seconds to find as many words as you can. You won't see the clock ticking, but each time you input a word, you'll see how many seconds you have left. As soon as time's run out, you'll be allowed to type one last word. (To quit the game early, hit ctrl-c.)

Okay, now go ahead and execute

```
~cs50/hacker3/scramble
```

again. Odds are the grid of letters changed? That's because the distribution code uses `rand`, a function that lets you generate pseudorandom numbers (and, thus, ASCII letters). But what if you don't want the grid's letters to change each time you run scramble, particularly while debugging? No problem, simply execute

```
~cs50/hacker3/scramble 1
```

to play grid #1, or

```
~cs50/hacker3/scramble 2
```

to play grid #2, and so forth. That (otherwise optional) command-line argument will be used as a "seed" for `rand` in order to perturb (*i.e.*, alter) its output.

☐ Okay, stop playing Scramble. Navigate your way to `~/Dropbox/hacker3` and open up `scramble.c` with `gedit`. (Remember how?) The challenge at hand is to complete this game's implementation. But first, let's take a tour.

Notice first that atop `scramble.c` are a bunch of constants. Take note of the comments above each. Recall that declaring as constants values that you intend to use multiple times throughout your code tends to be good practice, so that you can change the value as needed in a single place.

Next, below those constants are some global variables. Global variables tend to be frowned upon (because there's usually a cleaner way to achieve some goal). But when the sole purpose of a file is to implement some program, as is the case here with `scramble.c`, it's not unreasonable to use globals to avoid passing around particularly important values again and again among several functions. For instance, we've declared `grid` as a global simply because so many functions will need access to it anyway, as you'll eventually see.

Notice next how we've utilized `typedef` and `struct` to declare our very own data type called `word`, inside of which is a `bool` and an `array`. We'll use a whole bunch of those structures in order to keep track of the words in that dictionary you downloaded (and whether the user has found them on the grid).

Below `word`, meanwhile, is `dictionary`, which we've declared as a `struct` without using `typedef`. The result is that this program will have just one `dictionary` structure, inside of which is an `int` and an array of words (each of which is of type `word`).

Consider the prototypes below `dictionary` a sneak preview of the functions to come!

Incidentally, take care not to change any code related to `logfile`, which we use to automate some tests of your code!

☐ Okay, next read through `main`, focusing first on the comments and then on the code. If unsure at first glance what some line does, take some time to figure it out. It'll be a lot easier to write new code if you understand the code that's already there! If unfamiliar with some function, try to find it at `https://www.cs50.net/resources/cppreference.com/`, else consult its "man page." For instance, to pull up the manual for `atoi`, execute the below.[2]

```
man atoi
```

Notice, incidentally, how we're utilizing some "ANSI color codes" in main in order to output red text when the game's timer expires. They're a bit cryptic, to be sure, but pretty easy to use. See `http://pueblo.sourceforge.net/doc/manual/ansi_color_codes.html` for other colors.

Also, while debugging your program, you might want to comment out the call to `clear` in `main` so that you can see everything printed by `printf`, without anything disappearing.

☐ Next read through each of the functions below `main`. Don't fret if you don't understand `find` and `crawl`, but do take a stab at reading through them. It turns out that `crawl` implements a "recursive" algorithm (whereby `crawl` calls itself) that searches the grid horizontally, vertically, and diagonally for a particular word, "marking" letters temporarily as it visits them so that it doesn't accidentally get caught in an infinite loop.

Meanwhile, `initialize` might look a bit intimidating, but spend some time figuring out how it goes about initializing the grid with a distribution of letters. The man page for `rand` (albeit a bit cryptic itself) might help you figure out all the arithmetic.

Finally, load definitely has some new syntax, particularly `FILE`. We'll revisit `FILE` and more in the weeks to come. For now, know that `load` simply loads a whole bunch of words, one per line, from a file into an array.

Hm, it seems we forgot to implement `draw` and `lookup`. D'oh.

[2] On occasion, you may need to execute

```
man 2 function
```

or

```
man 3 function
```

where `function` is some function's name, lest you pull up the manual for a Linux command as opposed to a C function. For instance, the man page for C's `printf` is in (chapter) 3 and not 1, which is the default if you don't specify a chapter explicitly.

☐ Suffice it to say we need your help finishing this implementation of `scramble`! And just a couple other favors, too, if you don't mind!

- ☐ Complete the implementation of `draw` (using some loops and `printf`) in such a way that `grid[i][j]` represents the letter in row `i`, column `j`. You're welcome to stray from the aesthetics of the staff's own solution.
- ☐ Complete the implementation of `lookup` in such a way that the function returns true iff (*i.e.*, if and only if) `word` is in `dictionary`. Odds are you can do better than linear search!
- ☐ Enhance `scramble` in such a way that anytime the user types `INSPIRATION`, up to three words are displayed, one of length 5 (if any), one of length 4 (if any), and one of length 3 (if any), all of which are in the dictionary and in the grid but not yet found.
- ☐ By default, the distribution code is case-sensitive, whereby if `FOO` is in `dictionary`, the user must type `FOO`, not `foo`, in order to score. Alter main in such a way that the user can type `FOO` or `foo` (or even `FoO` or any other capitalization thereof) in order to score.
- ☐ Enhance `scramble` in such a way that letters in words found contribute the values below to a user's score (instead of the default of 1):

A = 1	G = 3	L = 2	Q = 10	V = 5
B = 4	H = 3	M = 4	R = 1	W = 4
C = 4	I = 1	N = 2	S = 1	X = 8
D = 2	J = 10	O = 1	T = 1	Y = 3
E = 1	K = 5	P = 4	U = 2	Z = 10
F = 4				

☐ Phew, now you can play (well, maybe after some debugging) your own version of `scramble`!

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.

☐ Open a terminal window (as via **Menu > Programming > Terminal** or within gedit) then execute

```
update50
```

to ensure you have the latest release of the appliance. Then execute:

```
cd ~/Dropbox/hacker3
```

And then execute:

```
ls
```

At a minimum, you should see `scramble.c`. If not, odds are you skipped some step(s) earlier! If you do see those files, you are ready to submit your source code to us. Execute:

```
submit50 ~/Dropbox/hacker3
```

and follow the on-screen instructions. That command will essentially upload your entire `~/Dropbox/hacker3` directory to CS50's servers, where your TF will be able to access it. The command will inform you whether your submission was successful or not. And you may inspect your submission at `cs50.net/submit`.

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 `cs50.net/discuss` 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 right away!

☐ Head to the URL below where a short form awaits:

```
https://www.cs50.net/psets/3/
```

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

This was Problem Set 3.