Problem Set 3: The Game of Fifteen

due by 7:00pm on Fri 10/1

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 9/29), 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.

- Introduce you to larger programs and programs with multiple source files.
- Empower you with Makefiles.
- Implement a party favor.

Recommended Reading.

- Section 17 of `http://www.howstuffworks.com/c.htm`.
- Chapters 20 and 23 of *Absolute Beginner's Guide to C*.
- Chapters 13, 15, and 18 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.

☐ SSH to `cloud.cs50.net` and execute the command below.[1]

```
cp -r ~cs50/pub/src/psets/pset3 ~
```

That command will copy the staff's `pset3/` directory, inside of which is some "distribution code," files (and subdirectories) that you'll need for this problem set, into your own home directory. The `-r` switch triggers a "recursive" copy. Navigate your way to your copy by executing the command below.

```
cd ~/pset3/
```

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.

```
fifteen/  find/
```

As this output implies, your work for this problem set will be organized within two subdirectories.

☐ Don't forget about `help.cs50.net`! Not only can you post questions of your own, you can also search for or browse answers to questions already asked by others. And never fear asking "dumb questions." Students' posts to the course's bulletin board are anonymized. Only the staff, not fellow students, will know who you are![2]

Be sure, too, to attend or watch this problem set's walkthrough, the video for which will be available at `http://www.cs50.net/psets/`.

And, of course, there are plenty of office hours: `http://www.cs50.net/ohs/`.

[1] Note that there's a space between `pset3` and that last tilde (`~`), the latter of which, recall, represents your home directory.
[2] Thus, only we will make fun.

Find.

☐ Now let's dive into the first of those subdirectories. Execute the command below.

```
cd ~/pset3/find/
```

If you list the contents of this directory, you should see the below.

```
helpers.c  helpers.h  Makefile  find.c  generate.c
```

Wow, that's a lot of files, eh? Not to worry, we'll walk you through them.

☐ Implemented in `generate.c` is a program that uses `rand` to generate a whole bunch of pseudorandom numbers, one per line. (Remember `rand` from Problem Set 1?) Go ahead and compile this program by executing the command below.

```
make generate
```

Now run the program you just compiled by executing the command below.

```
./generate
```

You should be informed of the program's proper usage, per the below.

```
Usage: generate n [s]
```

As this output suggests, this program expects one or two command-line arguments. The first, `n`, is required; it indicates how many pseudorandom numbers you'd like to generate. The second, `s`, is optional, as the brackets are meant to imply; if supplied, it represents the value that the pseudorandom-number generator should use as its seed. (Remember from Problem Set 1 what a seed is?) Go ahead and run this program again, this time with a value of, say, `10` for `n`, as in the below; you should see a list of 10 pseudorandom numbers.

```
./generate 10
```

Run the program a third time using that same value for `n`; you should see a different list of 10 numbers. Now try running the program with a value for `s` too (*e.g.*, `0`), as in the below.

```
./generate 10 0
```

Now run that same command again:

```
./generate 10 0
```

Bet you saw the same "random" sequence of ten numbers again? Yup, that's what happens if you don't vary a pseudorandom number generator's initial seed.

☐ Now take a look at `generate.c` itself with Nano. (Remember how?) Comments atop that file explain the program's overall functionality. But it looks like we forgot to comment the code itself. Read over the code carefully until you understand each line and then comment our code for us, replacing each `TODO` with a phrase that describes the purpose or functionality of the corresponding line(s) of code. Realize that a comment flanked with `/*` and `*/` can span lines whereas a comment preceded by `//` can only extend to the end of a line; the latter is a feature of C99 (the version of C that we've been using). If Problem Set 1 feels like a long time ago, you might want to read up on `rand` and `srand` again at the URLs below.

```
http://www.cs50.net/resources/cppreference.com/stdother/rand.html
http://www.cs50.net/resources/cppreference.com/stdother/srand.html
```

Or you can execute the commands below.

```
man rand
man srand
```

Once done commenting `generate.c`, re-compile the program to be sure you didn't break anything by re-executing the command below.

```
make generate
```

If `generate` no longer compiles properly, take a moment to fix what you broke!

Now, recall that `make` automates compilation of your code so that you don't have to execute `gcc` manually along with a whole bunch of switches. Notice, in fact, how `make` just executed a pretty long command for you, per the tool's output. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program, particularly when they involve multiple source (*i.e.*, `.c`) files. And so we'll start relying on "Makefiles," configuration files that tell `make` exactly what to do.

How did `make` know how to compile `generate` in this case? It actually used a configuration file that we wrote. Using Nano, go ahead and look at the file called `Makefile` that's in the same directory as `generate.c`. This `Makefile` is essentially a list of rules that we wrote for you that tells `make` how to build `generate` from `generate.c` for you. The relevant lines appear below.

```
generate: generate.c
    gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o generate generate.c
```

The first line tells `make` that the "target" called `generate` should be built by invoking the second line's command. Moreover, that first line tells `make` that `generate` is dependent on `generate.c`, the implication of which is that `make` will only re-build `generate` on subsequent runs if that file was modified since `make` last built `generate`. Neat time-saving trick, eh? In fact, go ahead and execute the command below again, assuming you haven't modified `generate.c`.

```
make generate
```

You should be informed that `generate` is already up to date. Incidentally, know that the leading whitespace on that second line is not a sequence of spaces but, rather, a tab. Unfortunately, `make` requires that commands be preceded by tabs, so be careful not to change them to spaces with Nano (which automatically converts tabs to four spaces), else you may encounter strange errors! The `-Werror` flag, recall, tells `gcc` to treat warnings (bad) as though they're errors (worse) so that you're forced (in a good, instructive way!) to fix them.

☐ Now take a look at `find.c` with Nano. Notice that this program expects a single command-line argument: a "needle" to search for in a "haystack" of values. Once done looking over the code, go ahead and compile the program by executing the command below.

```
make find
```

Notice, per that command's output, that Make actually executed the below for you.

```
gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o find find.c helpers.c -lcs50 -lm
```

Notice further that you just compiled a program comprising not one but two `.c` files: `helpers.c` and `find.c`. How did `make` know what to do? Well, again, open up `Makefile` to see the man behind the curtain. The relevant lines appear below.

```
find: find.c helpers.c helpers.h
    gcc -ggdb -std=c99 -Wall -Werror -Wformat=0 -o find find.c helpers.c -lcs50 -lm
```

Per the dependencies implied above (after the colon), any changes to `find.c`, `helpers.c`, or `helpers.h` will compel `make` to rebuild `find` the next time it's invoked for this target.

Go ahead and run this program by executing, say, the below.

```
./find 13
```

You'll be prompted to provide some hay (*i.e.*, some integers), one "straw" at a time. As soon as you tire of providing integers, hit ctrl-d to send the program an EOF (end-of-file) character. That character will compel `GetInt` from the CS50 Library to return `INT_MAX`, a constant that, per `find.c`, will compel find to stop prompting for hay. The program will then look for that needle in the hay you provided, ultimately reporting whether the former was found in the latter. In short, this program searches an array for some value.

In turns out you can automate this process of providing hay, though, by "piping" the output of generate into find as input. For instance, the command below passes 1,024 pseudorandom numbers to `find`, which then searches those values for 13.

```
./generate 1024 | ./find 13
```

Alternatively, you can "redirect" `generate`'s output to a file with a command like the below.

```
./generate 1024 > numbers.txt
```

You can then redirect that file's contents as input to `find` with the command below.

```
./find 13 < numbers.txt
```

Let's finish looking at that `Makefile`. Notice the line below.

```
all: find generate
```

This target implies that you can build both `generate` and `find` simply by executing the below.

```
make all
```

Even better, the below is equivalent (because `make` builds a Makefile's first target by default).

```
make
```

If only you could whittle this whole problem set down to a single command! Finally, notice these last lines in `Makefile`:

```
clean:
    rm -f *.o a.out core find generate
```

This target allows you to delete all files ending in `.o` or called `a.out`, `core` (tsk, tsk), `find`, or `generate` simply by executing the command below.

```
make clean
```

Be careful not to add, say, `*.c` to that last line in `Makefile`! (Why?) Any line, incidentally, that begins with `#` is just a comment.

☐ And now the fun begins! Notice that `find.c` calls `sort`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! Take a peek at `helpers.c` with Nano, and you'll see that `sort` returns immediately, even though `find`'s `main` function does pass it an actual array. To be sure, we could have put the contents of `helpers.h` and `helpers.c` in `find.c` itself. But it's sometimes better to organize programs into multiple files, especially when some functions (*e.g.*, `sort`) are essentially utility functions that might later prove useful to other programs as well, much like those in the CS50 Library.

Incidentally, recall the syntax for declaring an array. Not only do you specify the array's type, you also specify its size between brackets, just as we do for `haystack` in `find.c`:

```
int haystack[HAY_MAX];
```

But when passing an array, you only specify its name, just as we do when passing `haystack` to `sort` in `find.c`:

```
sort(haystack, size);
```

(Why do we also pass in the size of that array separately?)

When declaring a function that takes a one-dimensional array as an argument, though, you don't need to specify the array's size, just as we don't when declaring `sort` in `helpers.h` (and `helpers.c`):

```
void sort(int values[], int n);
```

Go ahead and implement `sort` so that the function actually sorts, from smallest to largest, the array of numbers that it's passed, in such a way that its running time is in $O(n^2)$, where n is the array's size. Odds are you'll want to implement Bubble Sort or Selection Sort, if only because we discussed them in Week 3. Just realize that there's no one "right" way to implement either of those algorithms; variations abound. In fact, you're welcome to improve upon them as you see fit, so long as your implementation remains in $O(n^2)$. However, take care not to alter our declaration of `sort`. Its prototype must remain:

```
void sort(int values[], int n);
```

As this return type of `void` implies, this function must not return a sorted array; it must instead "destructively" sort the actual array that it's passed by moving around the values therein. As we'll discuss in Week 4, arrays are not passed "by value" but instead "by reference," which means that `sort` will not be passed a copy of an array but, rather, the original array itself.

We leave it to you to determine how to test your implementation of `sort`. But don't forget that `printf` and, per Week 3's second lecture, `gdb` are your friends. And don't forget that you can generate the same sequence of pseudorandom numbers again and again by explicitly specifying `generate`'s seed. Before you ultimately submit, though, be sure to remove any such calls to `printf`, as we like our programs' outputs just they way they are!

Incidentally, check out **Resources** on the course's website for a great little quick-reference guide for `gdb`. If you'd like to play with the staff's own implementation of `find` on `cloud.cs50.net`, you may execute the below.

```
~cs50/pub/solutions/pset3/find
```

☐ Need help? Head to `help.cs50.net`!

☐ Now that `sort` (presumably) works, it's time to improve upon `search`, the other function that lives in `helpers.c`. Notice that our version implements "linear search," whereby `search` looks for `value` by iterating over the integers in `array` linearly, from left to right. Rip out the lines that we've written and re-implement `search` as binary search, that divide-and-conquer strategy that

we employed in Week 0 in order to search through phone book.[3] You are welcome to take an iterative or a recursive approach. If you pursue the latter, though, know that you may not change our declaration of `search`, but you may write a new, recursive function (that perhaps takes different parameters) that `search` itself calls.

The Game Begins.

☐ And now it's time to play. The Game of Fifteen is a puzzle played on a square, two-dimensional board with numbered tiles that slide. The goal of this puzzle is to arrange the board's tiles from smallest to largest, left to right, top to bottom, with an empty space in board's bottom-right corner, as in the below.[4]

Sliding any tile that borders the board's empty space in that space constitutes a "move." Although the configuration above depicts a game already won, notice how the tile numbered 12 or the tile numbered 15 could be slid into the empty space. Tiles may not be moved diagonally, though, or forcibly removed from the board.

Although other configurations are possible, we shall assume that this game begins with the board's tiles in reverse order, from largest to smallest, left to right, top to bottom, with an empty space in the board's bottom-right corner. If, however, and only if the board contains an odd number of tiles (*i.e.*, the height and width of the board are even), the positions of tiles numbered 1 and 2 must be swapped, as in the below.[5] The puzzle is solvable from this configuration.

[3] No need to tear anything in half.
[4] Figure from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.
[5] Figure adapted from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.

☐ Navigate your way to `~/pset3/fifteen/`, and take a look at `fifteen.c` with Nano. Within this file is an entire framework for The Game of Fifteen. The challenge up next is to complete this game's implementation.

But first go ahead and compile the framework. (Can you figure out how?) And, even though it's not yet finished, go ahead and run the game. (Can you figure out how?)

Phew. It appears that the game is at least partly functional. Granted, it's not much of a game yet. But that's where you come in.

Read over the code and comments in `fifteen.c` and then answer the questions below in `questions.txt`.

i. Besides 4 × 4 (which are The Game of Fifteen's dimensions), what other dimensions does the framework allow?
ii. With what sort of data structure is the game's board represented?
iii. What function is called to greet the player at game's start?
iv. What functions do you apparently need to implement?

☐ Alright, get to it, implement this game. Remember, take "baby steps." Don't try to bite off the entire game at once. Instead, implement one function at a time and be sure that it works before forging ahead. In particular, we suggest that you implement the framework's functions in this order: `init`, `draw`, `move`, `won`. Any design decisions not explicitly prescribed herein (*e.g.*, how much space you should leave between numbers when printing the board) are intentionally left to you. Presumably the board, when printed, should look something like the below, but we leave it to you to implement your own vision.

```
15 14 13 12

11 10  9  8

 7  6  5  4

 3  1  2  _
```

Incidentally, recall that the positions of tiles numbered 1 and 2 should only start off swapped (as they are in the 4 × 4 example above) if the board has an odd number of tiles (as does the 4 × 4 example above). If the board has an even number of tiles, those positions should not start off swapped. And so they do not in the 3 × 3 example below:

```
8  7  6

5  4  3

2  1  _
```

To test your implementation of `fifteen`, you can certainly try playing it. (Know that you can force your program to quit by hitting ctrl-c.) Be sure that you (and we) cannot crash your

program, as by providing bogus tile numbers. And know that, much like you automated input into `find`, so can you automate execution of this game. In fact, in `~cs50/pub/tests/pset3/` are `3x3.txt` and `4x4.txt`, winning sequences of moves for a 3 × 3 board and a 4 × 4 board, respectively. To test your program with, say, the first of those inputs, execute the below.

```
./fifteen 3 < ~cs50/pub/tests/pset3/3x3.txt
```

Feel free to tweak the appropriate argument to `usleep` to speed up animation. In fact, you're welcome to alter the aesthetics of the game. For (optional) fun with "ANSI escape sequences," including color, take a look at our implementation of clear and check out the URL below for more tricks.

```
http://isthe.com/chongo/tech/comp/ansi_escapes.html
```

You're welcome to write your own functions and even change the prototypes of functions we wrote. But we ask that you not alter the flow of logic in `main` itself so that we can automate some tests of your program once submitted. In particular, `main` must only returns `0` if and when the user has actually won the game; non-zero values should be returned in any cases of error, as implied by our distribution code. If in doubt as to whether some design decision of yours might run counter to the staff's wishes, simply contact your teaching fellow.

If you'd like to play with the staff's own implementation of `fifteen` on `cloud.cs50.net`, you may execute the below.

```
~cs50/pub/solutions/pset3/fifteen
```

If you'd like to see an even fancier version, one so good that it can play itself, try out our solution to the Hacker Edition by executing the below.

```
~cs50/pub/solutions/hacker3/fifteen
```

Instead of typing a number at the game's prompt, type `GOD` instead. Neat, eh?[6]

☐ `help.cs50.net`!

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 9/29.

☐ SSH to `cloud.cs50.net`, if not already there, and then execute:

```
cd ~/pset3
```

[6] To be clear, implementation of God Mode is part of this problem set's Hacker Edition. You don't need to implement God Mode for this standard edition! But it's still pretty neat, eh?

If informed that there is "no such file or directory," odds are you skipped some step(s) earlier! Not a problem, we'll help you fix in a moment. If the command indeed works, proceed to execute this command next:

```
ls
```

At a minimum, you should see `fifteen` and `find`. If not, odds are you skipped some more steps earlier! If you do see those files, you're ready to submit your source code to us:[7]

```
~cs50/pub/bin/submit pset3
```

That command will essentially copy your entire `~/pset3` directory into CS50's own account, where your TF will be able to access it. 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. (The `submit` program's a bit old and annoyingly flakey.) But if informed that your "work HAS been submitted," it indeed has.

Now, if you instead see some message informing you that your code is not where it should be, you'll need to fix that before you submit. Review the last two pages of Problem Set 1's spec for some tips!

☐ Anytime after lecture on Wed 9/29 but before this problem set's deadline, head to the URL below where a short form awaits:

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

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 3.

[7] Note that there is no slash between this tilde (`~`) and `cs50`.