Computer Science 50: Introduction to Computer Science I
Harvard College
Fall 2008

Problem Set 3: The Game of Fifteen

due by 7:00 P.M. on Friday, 17 October 2008

Goals.

- Introduce you to larger programs and programs with multiple source files.
- Empower you with Makefiles and RCS.
- Acquaint you with pseudorandom numbers.
- Play (but call it work).

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 (*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 course 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.

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.

You may even 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.

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.

0. For perhaps the last time, we're again going to have you code on `nice.fas.harvard.edu` for this problem set.[1] SSH to `nice.fas.harvard.edu` and execute the command below.

   ```
   cp -r ~cs50/pub/distributions/pset3/ ~/cs50/
   ```

   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 `~/cs50/` directory. The `-r` switch triggers a "recursive" copy. Navigate your way to your copy by executing the command below.

   ```
   cd ~/cs50/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/  questions.txt
   ```

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

1. Don't forget that the course's website has a bulletin board! 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]

2. Speaking of questions, head on over to

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

   for some quick ones from us!

[1] Note that the Hacker Edition of this problem set has students instead SSH to `hacker3.cs50.net`, an updated prototype of our new cluster "in the cloud."
[2] Thus, only we will make fun. `0:-)`

Find.

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

   ```
   cd ~/cs50/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.

4. Implemented in `generate.c` is a "pseudorandom-number generator" (PRNG), a program that outputs a whole bunch of random numbers, one per line. Actually, these numbers are generated not so much randomly as they are "pseudorandomly." Because a computer is a deterministic device (*i.e.*, it can only do what it's told to do), it can't just pick a number off the top of its head. However, algorithms exist that enable a computer to generate sequences of numbers that appear to be random in the sense that there's no obvious pattern to them. C provides a function called `rand()` for exactly this purpose. The language also provides a function called `srand()` that is used to "seed" the PRNG. To "seed" a PRNG means to feed an initial value, *s*, to its generating algorithm, *g*. Typically, the first number returned by a PRNG is $g(s)$; the second is $g(g(s))$; the third is $g(g(g(s)))$; and so forth. Hence, you can generate the same sequence of "random numbers" simply by seeding the PRNG with the same initial value. The current time, often measured in seconds since some particular moment in the past, is typically used as the seed to a PRNG so that the seed is not hard-coded into a program but instead dynamic.

   Anyhow, go ahead and compile this program by executing the command below.

   ```
   gcc –ggdb -std=c99 -Wall -o generate generate.c
   ```

   Wow, that's quite the command, eh? It turns out you've been executing commands like that one all along. Prior to this problem set, anytime you typed

   ```
   gcc
   ```

   it was as though you were typing

   ```
   gcc –ggdb -std=c99 -Wall
   ```

   because we had "aliased" the former command to the latter to save you keystrokes and avoid confusion. (Remember `cs50setup`? That's one of the things it did for you.) If curious, the `-ggdb` switch tells gcc to include "debugging symbols" in your binaries to facilitate debugging with GDB. The `-std=cs99` switch tells GCC that your code might include C99 syntax. And the `-Wall` switch tells GCC to report all possible warnings anytime it detects possible problems with your code.

It's time for some training wheels to come off, though! We've thus deactivated that alias. But we don't expect you to start typing ridiculously long commands. We'll just start using `make` even more. For now, though, go ahead and 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 the generate. The second, `s`, is optional, as implied by the brackets; if supplied, it represents the value that the pseudorandom-number generator should use as its seed. 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 twice more, still using that same value for `n`, but this time also providing some value for `s` both times, as in the below; the output of both executions should be identical.

```
generate 10 0
```

Think of this last command, with its seed of 0, as having generated the PRNG's 0th possible sequence of 10 pseudorandom numbers.

5. 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, recall, is a feature of C99. If curious about `rand` and `srand`, pull up the URLs below.

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

Or execute the commands below.

```
man 3 rand
man 3 srand
```

Note that if you instead execute the command below, you'll pull up the man page for a program (not function) named rand in section 1 of the Linux Programmer's Manual.

```
man rand
```

Functions, by contrast, tend to be documented in sections 2 and 3. To avoid any ambiguity, then, you sometimes need to tell man the section you want. If curious as to what's where, execute, believe it or not, the command below.

```
man man
```

Once done commenting `generate.c`, re-compile the program to be sure you didn't break anything. Rather than execute that long command from earlier, though, simply execute the command below.

```
make generate
```

As you may recall from Problem Set 1, `make` automates compilation of your code. 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` can't do absolutely everything for you; 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? 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 -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, else you may encounter strange errors!

6. 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 -o find helpers.c find.c -lcs50
```

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

Per the dependencies implied above (after the colon), any changes to `helpers.c`, `helpers.h`, or `find.c` 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 CS 50's library to return `INT_MAX`, which, 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: generate find
```

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 generate find
```

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

7. Phew, lots of good stuff so far, and it's almost time to start coding. But one last lesson for you. From personal (traumatic) experience, you probably already know that backups are a good thing. What you might not know is that a number of Linux tools exist to facilitate the process of backing up source code. Starting with this problem set, you'll want to use a utility called RCS (Revision Control System) to make regular, incremental backups of your source code. Not only will RCS enable you to restore your most recently backed-up copy of a file in the event of trauma, it will also enable you to restore different versions of your source code, in the event you realize that the code you wrote a few days ago was much better than what you've been producing since.

   Go ahead and "check in" (*i.e.*, backup) your initial version of `find.c` by executing the command below.

   ```
   ci find.c
   ```

   You'll be prompted for a description for this file. Go ahead and describe the purpose of this file in a few words, then enter `.` or hit ctrl-d on a line of its own to save the description. You should be informed that version 1.1 of this file, your "initial revision," has been checked in. If you list the contents of your current working directory, you'll notice that you now have a directory called `RCS` therein, inside of which is `find.c,v`, which is where RCS (*i.e.*, `ci`) records changes to your file.

   Henceforth, anytime you want to check in your latest version of `find.c`, simply execute the same command as before, per the below.

   ```
   ci find.c
   ```

No longer will you be prompted for a description but, rather, a "log message," which is even more important than the file's initial description. Log messages are supposed to help you remember what's different between this version and your last (*e.g.*, "I changed my `while` loop to a `do-while` loop"). Entering that message might be tedious, but, trust us, you're not going to remember what was special about version 1.9 at 3:00 A.M. without a little help. As before, enter . or hit ctrl-d on a line of its own to save your message. Each time you check in a newer version of your file, RCS will assign an appropriate version number.

Suppose, for future reference, that you want to restore, say, version 1.1 of `find.c`. If you don't want to clobber (*i.e.*, overwrite) your current version, be sure to check it in first! Then proceed to execute the command below.

```
co –r1.1 find.c
```

You should find that version 1.1 of find.c has been restored to your current working directory. So that you know which version of `find.c` is which, execute the command below to see your own log messages.

```
rlog find.c
```

To save time, know that you can check in multiple files at once, as with the command below.

```
ci *.c *.h
```

For disk space's sake, RCS will allow you to check in source files on `nice.fas.harvard.edu` but not binaries.

8. 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 CS 50's own 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` using any algorithm that's in $O(n^2)$ so that the function actually sorts, from smallest to largest, any array of integers that it's passed.[3] You may not alter the function's declaration. In particular, its return type must remain `void`. Rather than return a new, sorted array, then, the function must instead "destructively" sort the actual array that it's passed.

Don't forget to check in `helpers.c` before making your changes! (Remember how?)

We leave it to you to determine how to test your implementation of `sort`. But don't forget that `printf` and, now, `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 `nice.fas.harvard.edu`, you may execute the below.

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

9. Bulletin board!

10. Now that `sort` (presumably) works, you can improve upon `search`. Notice that our version implements linear search. Rip out those lines that we've written and re-implement search as binary search!

11. Despite your enhancements to sort and search, you may find that your version of `find`, once built with your changes, is now slower than ours. But why? Explain in a sentence or more in `pset3/questions.txt` why your "new and improved" code is slower than ours. In another sentence or more, explain why one might ever want to bother sorting then searching with binary search.

[3] You're welcome to turn to Chapter 23 of *Absolutely Beginner's Guide to C* for guidance, but we suggest that you instead allow yourself only Week 3's pseudocode.

The Game Begins.

12. 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 into that 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.

13. Navigate your way to `~/cs50/pset3/fifteen/`, and take a look at `fifteen.c` with Nano. Within this file is the entire framework for The Game of Fifteen (and variants thereof). The challenge ultimately at hand is to complete this game's implementation.

[4] Figure from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.
[5] Figure adapted from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.

But first check in `fifteen.c`! (Remember how?) Then 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 `pset3/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?
v. Have you actually played and won The Game of Fifteen in real life?

14. 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
```

To test your implementation, 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 the 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
```

But we ask that you not alter the flow of logic in main 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 `nice.fas.harvard.edu`, 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]

15. Bulletin board!!

Submitting Your Work.

16. Ensure that your work is in `~/cs50/pset3/`. Submit your work by executing the command below.

```
cs50submit pset3
```

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.

[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?