NOTICE TO PODCAST SUBSCRIBERS

The source code for CS 50's library as well as for Fall 2007's problem sets in general can be found at `http://cs50.tv/`.

Problem Set 3: The Game of Fifteen

out of 41 points

due by 7:00 P.M. on Saturday, 27 October 2007

This problem set has more pages than past problem sets, but most contain narrative. Even so, best to start early, as the programs we're writing are growing in size.

Goals.

The goals of this problem set are to:

- Introduce you to larger programs and programs with multiple source files.
- Empower you with Make and RCS.
- Acquaint you with pseudorandom numbers.
- Play in God Mode.

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

- Section 16 of `http://www.howstuffworks.com/c.htm`.
- Chapters 13, 15, and 18 of *Programming in C*.

C99.

Starting with this problem set, you are welcome (but not expected or required) to use features of C99, a version of C newer than that in which most lectures' source code has been written. Thanks to C99, you can now prefix one-line comments with just `//`, and you needn't declare all of your variables at the very top of your functions, and more. For even more bells and whistles, follow the link to **A Tour of C99** under **Resources** on the course's website.

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 class 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.

All forms of cheating will be dealt with harshly.

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.

Grading Metrics.

Each question is worth the number of points specified parenthetically in line with it.

Your responses to questions requiring exposition will be graded on the basis of their clarity and correctness. Your responses to questions requiring code will be graded along the following axes.

> **Correctness.** To what extent does your code adhere to the problem's specifications?
> **Design.** To what extent is your code written clearly, efficiently, elegantly, and/or logically?
> **Style.** To what extent is your code commented and indented, your variables aptly named, *etc.*?

Rest assured that grades will be normalized across sections at term's end.

Getting Started.

0. SSH to `nice.fas.harvard.edu` and execute the command below.

```
cp -r ~cs50/pub/ps/distributions/ps3/ ~/cs50/
```

That command will copy the staff's `ps3` directory, inside of which are several subdirectories and files 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/ps3/
```

If you list the contents of your current working directory, 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. Let's get started.

1. (6 points.) Look up your computer's "specs" online.[1] In other words, figure out your desktop's or laptop's make and model (presumably by reading both off some sticker or the case itself), surf on over to Google or the manufacturer's website, and find your computer's technical specifications. Then, in `ps3/questions.txt`, tell us six or more of the below.

   i. Your computer's make and model.
   ii. The make, model, and speed of your computer's CPU.
   iii. The amount of L1 cache in your computer.
   iv. The amount of L2 cache in your computer.
   v. The amount of RAM in your computer.
   vi. The size of your computer's hard drive.
   vii. The number of keys on your keyboard.

[1] If you haven't your own computer, simply answer these questions for someone else's computer. The experience will be the same!

Find.

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

```
cd 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.

3. 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 pseudorandom-number generator. To "seed" a generator means to feed an initial value, s, to its generating algorithm, g. Typically, the first number returned by such a generator 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 generator 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 generator 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 soon equip you with a better tool. 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.

4. (2 points.) Now take a look at `generate.c` itself. 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, again, is a feature of C99. If curious about `rand` and `srand`, pull up the URLs below.

```
http://www.cppreference.com/stdother/rand.html
http://www.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
```

Henceforth, we'll be using Make, a tool that "controls the generation of executables and other non-source files of a program from the program's source files." In other words, Make automates compilation of your code. Notice, in fact, how Make just executed that long command for you, per the tool's output.

How did Make know what to do? Go ahead and look at `Makefile`. 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 your text editor, else you may encounter strange errors!

5. Now take a look at `find.c`. 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, 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`, `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.

6. 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.

7. (9 points.) 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`, and you'll see that `sort` returns immediately, even though `find`'s `main` function does pass it an actual array. Notice, incidentally, the syntax for passing an 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.

   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 $O(n)$, where n is the

array's size.[2] You may assume that each of the arrays' numbers will be non-negative and less than `LIMIT`, a constant defined in `generate.c`. But the array might contain duplicates.

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.

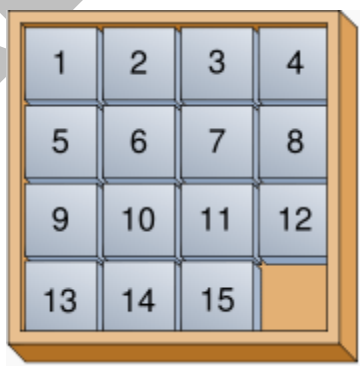
Don't forget to check in `helpers.c` before making your changes!

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!

8. (9 points.) 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!

The Game Begins.

9. 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. Though, if you don't know already this, know that we've missed you at the past few lectures. 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.[3]

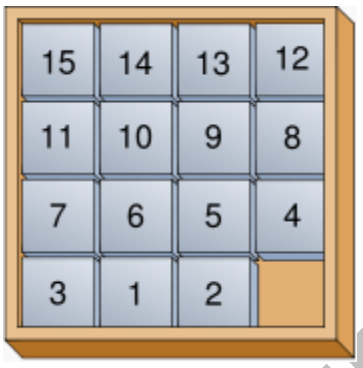


Sliding any tile that borders the board's empty space into that constitutes a "move." Although the configuration above depicts a game already won, notice how the tile numbered 12 or 15

[2] Technically, because we've bounded with a constant the amount of hay that `find` will accept (and because the value of `sort`'s second parameter is bounded by an `int`'s 32 bits) the running time of `sort`, however implemented, is arguably $O(1)$. Even so, for the sake of this asymptotic challenge, think of the size of `sort`'s input as n.
[3] Figure from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.

could be slid into the empty space. Tiles may not be moved diagonally, though, or forcibly removed from the board.

Depicted below is just one configuration from which the puzzle is solvable.[4]



However, not all configurations lead to solutions. For the mathematics behind this puzzle's solvability, see the URL below.

```
http://mathworld.wolfram.com/15Puzzle.html
```

So long as you start from the winning configuration, though, and randomly slide one or more tiles one or more times, you will certainly reach a configuration from which the puzzle is solvable.

10. (15 points.) Navigate your way to `~/cs50/ps3/fifteen/`, and take a look at `Makefile` and `fifteen.c`. Within the latter is the entire framework for The Game of Fifteen (and variants thereof).

    Implement God Mode for this game.

    First implement `init` in such a way that the board is initialized to a pseudorandom but solvable configuration. Then complete the implementation of `draw`, `move`, and `won` so that a human can actually play the game. But embed in the game a cheat, whereby, rather than typing an integer betwen 1 and $d^2 - 1$, where d is the board's height and width, the human can also type

    ```
    GOD
    ```

    to compel "the computer" to take control of the game and solve it (using any strategy, optimal or non-optimal), making, say, only four moves per second so that the human can actually watch. Presumably, you'll need to swap out `GetInt` for something more versatile.

    To test your implementation, you can certainly try playing it yourself, with or without God Mode enabled. (Know that you can quit your program by hitting ctrl-c.) Be sure that you

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

(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 via input redirection if you store in some file a winning sequence of moves for some configuration.

You're welcome to alter the aesthetics of this game. 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
```

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 do ask that you try not to alter the overall flow of logic in `main` so that we can reliably automate testing of your program once submitted. If in doubt as to whether some design decision of yours might run counter to the staff's wishes, simply contact your teaching fellow.

Submitting Your Work.

11. Ensure that your work is in `~/cs50/ps3/`. Submit your work by executing the command below.

    ```
    cs50submit ps3 hacker
    ```

    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.