Problem Set 3: The Game of Fifteen

due by noon on Thu 9/29

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.

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.
- Introduce you to literature in computer science.
- 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*.

diff pset3.pdf hacker3.pdf.

- Hacker Edition dares you to implement `sort` in $O(n)$ instead of $O(n^2)$.
- Hacker Edition asks you to play God.

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 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 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 *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.

Grades.

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.

Getting Started.

☐ Welcome back!

Launch VirtualBox (as by double-clicking its icon wherever it's installed), and then boot the appliance (as by single-clicking it in VirtualBox's lefthand menu, and then clicking **Start**).

Upon reaching John Harvard's desktop, open a terminal window (remember how?) and type the below, followed by Enter:

```
sudo yum -y update
```

Input **crimson** if prompted for John Harvard's password. For security, <u>you won't see any characters as you type</u>. Realize that updating the appliance in this manner requires Internet access. If on a slow connection (or computer), it might take a few minutes to update the appliance. Don't worry if the process seems to hang if it decides to update a "package" called `cs50-appliance`; that one can take several minutes.

If you see messages like **Couldn't resolve host** or **Cannot retrieve metalink for repository**, those simply mean that the appliance doesn't currently have Internet access. Sometimes that happens if you've just awakened your computer from sleep or perhaps changed from wireless to wired Internet or vice versa. If your own computer does have Internet access (which you can confirm by trying to visit some website in a browser on your own computer) but the appliance does not (which you can confirm by trying to visit the same with Firefox within the appliance), try restarting the appliance (as by clicking the green icon in its bottom-right corner, then clicking **Restart**).[1] If, upon restart, the appliance still doesn't have Internet access, head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` for help!

Once the appliance has been updated, you should see **Complete!** in your terminal window. If there was nothing to update, you'll see **No packages marked for Update** instead.

☐ Just to be sure that everything worked, go ahead and execute that very same command again in a terminal window (though not while its first invocation is still running):

```
sudo yum -y update
```

Again, input **crimson** if prompted for John Harvard's password. (If only a few minutes have passed since the last update, you might not even be prompted.) You should now see **No packages marked for Update**, which means that your appliance is now up-to-date! If you see some error instead, try once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

[1] Alternatively, you can try typing:
```
sudo service network restart
```
But if that doesn't work, best to restart the appliance.

☐ 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!

Let's get you started. Go ahead and 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

```
cd
```

to return to your home directory, if not already there, and then execute

```
git clone http://cdn.cs50.net/2011/fall/psets/3/hacker3.git/
```

to download this problem set's distro into your appliance. You should see **`Cloning into hacker3...`** and then your prompt again. If you instead see **`fatal`** followed by **`not found: did you run`**, odds are you made a typo. Best to try again!

Once successful, you should find that you have a brand-new `hacker3` directory inside of your home directory. You can confirm as much with:

```
ls
```

So, what was that `git` command? `git` is a popular "distributed version control system," a tool that programmers use to download someone else's code, to maintain multiple versions of their own code, and to save their own code to a remote server. That last feature should sound familiar. Each time you run `submit50`, you're saving your code to CS50's server. In fact, all this time, `submit50` has been secretly using `git` to do just that! But more on `git` another time. For now, all we've used it for is to download (*i.e.*, "clone") this problem set's distro code into your home directory.[2]

Okay, now execute

```
cd ~/hacker3
```

to move yourself into (*i.e.*, open) the directory you just cloned. Your prompt should now resemble the below.

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

And if you execute

```
ls
```

[2] If already familiar with `git`, you're welcome to use it for local commits.

you should see

```
fifteen  find
```

which are two directories inside of which your programs will soon live! Fun times ahead!

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

The Real World.

☐ If you've dropped by office hours of late, particularly on a Tuesday or Wednesday, you'll know that having 100 or more laptops in the same room (not to mention nearly as many smart phones in pockets) tends to slow down (or take down altogether) wireless Internet access (otherwise known as 802.11 or Wi-Fi). In fact, visit most any hotel that has Wi-Fi, and you'll likely find that it slows down significantly at night, once folks have returned to their rooms.

Head on over to

```
http://en.wikipedia.org/wiki/Wi-Fi
```

and read up on how Wi-Fi itself works. No need to absorb every little detail, but odds are we'll have some questions for you before long!

Incidentally, HUIT is in the process of installing more Wi-Fi hardware (otherwise known as access points or APs) in the dining halls of Pfoho, Leverett, Quincy, and Lowell for us, so a better experience is on the way!

Find.

☐ Okay, let's dive into the first of those subdirectories. Execute the command below in a terminal window in your appliance.

```
cd ~/hacker3/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 a "pseudorandom-number generator" (via a function called `rand`) to generate a whole bunch of random (well, pseudorandom, since computers can't really generate truly random) numbers, one per line.[3] 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." A seed is simply an input to a pseudorandom-number generator that influences its outputs. For instance, if you seed `rand` by first calling `srand` (another function whose purpose is to "seed" `rand`) with an argument of, say, `1`, and then call `rand` itself three times, `rand` might return `17767`, then `9158`, then `39017`.[4] But if you instead seed `rand` by first calling `srand` with an argument of, say, `2`, and then call `rand` itself three times, `rand` might instead return `38906`, then `31103`, then `52464`. But if you re-seed `rand` by calling `srand` again with an argument of `1`, the next three times you call `rand`, you'll again get `17767`, then `9158`, then `39017`! See, not so random.

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 gedit. (Remember how?) Comments atop that file explain the program's overall functionality. But it looks like we forgot to comment the code itself.

[3] `https://www.cs50.net/resources/cppreference.com/stdother/rand.html`
[4] `https://www.cs50.net/resources/cppreference.com/stdother/srand.html`

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. For more details on `rand` and `srand`, recall that you can execute:

```
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 gedit, 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 gedit (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 gedit. 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 -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 -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
```

Note that, when piping output from `generate` into `find` in this manner, you won't actually see `generate`'s numbers, but you will see `find`'s prompts.

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 gedit, 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`:

```
if (!sort(haystack, size))
{
    printf("Could not sort haystack.\n");
    return 2;
}
```

(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`):

```
bool 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*), where *n* is the array's size.[5] Yes, this running time is possible, hax0r, because you may assume that each of the arrays' numbers will be non-negative and less than `LIMIT`, a constant defined in `generate.c`.[6] However, realize that the array might contain duplicates. Take care, though, not to alter our declaration of `sort`. Its prototype must remain:

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

As this return type of `bool` 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, thereafter returning `true` if and only if sorting was successful. (Odds are it will always be successful, but, if you employ certain tricks, it's possible to, say, run out of memory.) 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.

Although you may not alter our declaration of `sort`, you're welcome to define your own function(s) in `helpers.c` that sort itself may then call.

We leave it to you to determine how to test your implementation of `sort`. But don't forget that `printf` and, per Week 4's first 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!

[5] 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*.
[6] Leverage that assumption!

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` in the appliance, you may execute the below.

```
~cs50/hacker3/find
```

As always, do run

```
submit50 ~/hacker3
```

from time to time in order to back up your code to CS50's servers!

☐ 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.[7] You are welcome to take an iterative or, per Week 4, 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.

Again, do run

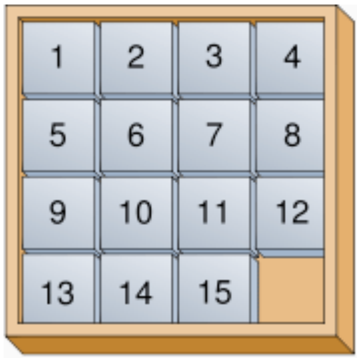
```
submit50 ~/hacker3
```

from time to time in order to back up your code to CS50's servers!

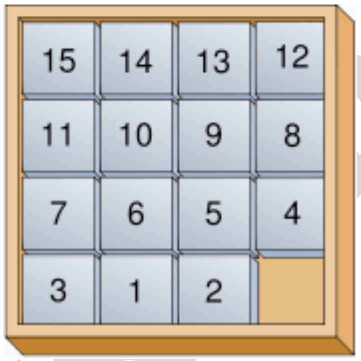
The Game Begins.

☐ 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.[8]

[7] No need to tear anything in half.
[8] Figure from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.



Sliding any tile that borders the board's empty space into 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. Depicted below is just one configuration from which the puzzle is solvable.[9]



☐ Navigate your way to `~/hacker3/fifteen/`, and take a look at `fifteen.c` with gedit. Within this file 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.[10] 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. It's fine if your implementation of God Mode only works (bearably fast) for `d ≤ 4`; you need not worry about

[9] Figure adapted from `http://en.wikipedia.org/wiki/Fifteen_puzzle`.
[10] To be clear, whereas the standard edition of this problem set requires that the board be initialized to a specific configuration, this Hacker Edition requires that it be initialized to a pseudorandom but still solvable configuration.

testing God Mode for `d > 4`. Oh and you can't implement God Mode by remembering how `init` initialized the board (as by remembering the sequence of moves that got your program to some pseudorandom but solvable state). That'd be, um, cheating. At cheating.

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 (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.

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 (albeit pseudorandom), 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 be 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 be swapped. Consider, for instance, the 3 × 3 example below:

```
8  7  6

5  4  3

2  1  _
```

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` so that we can automate some tests of your program. 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`, including God Mode, you may execute the below.

```
~cs50/hacker3/fifteen
```

Speaking of God Mode, where to begin? Well, first read up on this Game of Fifteen. Wikipedia is probably a good starting point:

```
http://en.wikipedia.org/wiki/N-puzzle
```

Then dive a bit deeper, perhaps reading up on an algorithm called A*.

```
http://en.wikipedia.org/wiki/A*_search_algorithm
```

Consider using "Manhattan distance" (aka "city-block distance") as your implementation's heuristic. If you find that A* takes up too much memory (particularly for `d ≥ 4`), though, you might want to take a look at iterative deepening A* (IDA*) instead:

```
http://webdocs.cs.ualberta.ca/~tony/RecentPapers/pami94.pdf
```

The staff's own implementation, meanwhile, utilizes an algorithm like that in this paper:

```
http://larc.unt.edu/ian/pubs/saml.pdf
```

You're welcome to expand your search for ideas beyond those in these papers, but take care that your research does not lead you to actual code. Curling up with others' pseudocode is fine, but please click away if you stumble upon actual implementations (whether in C or other languages).

Alright, get to it, implement this game!

☐ As before, do run

```
submit50 ~/hacker3
```

from time to time in order to back up your code to CS50's servers!

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.

☐ Recall that you obtained a CS50 Cloud account (*i.e.*, username and password) for Problem Set 1. If you don't remember your username and/or password, head to `https://cloud.cs50.net/` look up the former and/or change the latter. You'll be prompted to log in with your HUID (or XID) and PIN.

☐ Just in case we updated `submit50` since you started this problem set, open a terminal window and execute the below, inputting **crimson** if prompted for John Harvard's password.

```
sudo yum -y update
```

If there was something to update, you should see **Complete!** after a few seconds or minutes. If there was nothing to update, you should instead see **No packages marked for Update**. If you see any errors, try the command once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ To actually submit, first open a terminal window and execute:[11]

```
cd ~/hacker3
```

Then execute:

```
ls
```

At a minimum, you should see `fifteen` and `find`. If not, odds are you skipped some more steps earlier! If everything is as it should be, you are ready to submit your source code to us. Execute:[12]

```
submit50 ~/hacker3
```

When prompted for **Course**, input **cs50**; when prompted for **Repository**, input **hacker3**. When prompted for a username and password, input your CS50 Cloud username and password. For security, you won't see your password as you type it. That command will essentially upload your entire `~/hacker3` directory to CS50's repository, where your TF will be able to access it. The command will inform you whether your submission was successful or not. If provided with the URL of a PDF of your code (which further confirms its submission), right-click (or ctrl-click) the link, then choose **Open Link** from the menu that appears to open the PDF in Document Viewer.

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 `help.cs50.net` 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 within minutes!

☐ 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.

[11] Unless you decided to use `dropbox.com` and stored your files in, say, `~/Dropbox/hacker3`.
[12] *Ibid.*