Problem Set 3: Game of Fifteen

due Thu 10/3 at noon

By default, this problem set is due by Thu 10/3 at noon. However, if you

- **complete Searching before Wed 10/2 at noon (i.e., early),**
- **generate a coupon code confirming as much before Wed 10/2 at noon, and**
- **submit that coupon code at https://www.cs50.net/coupons/3 (https://www.cs50.net/coupons/3) before Wed 10/2 at noon,**

your deadline for the rest of the problem set's problems will be extended from Thu 10/3 at noon until Fri 10/4 at noon. If you do not submit a coupon code before Wed 10/2 at noon, this problem set, including Searching, is still due by Thu 10/3. In other words, by tackling part of the problem set early, you can extend your deadline for the rest!

Questions? Head to cs50.net/discuss (https://www.cs50.net/discuss) or join classmates at office hours (https://www.cs50.net/ohs)!

Objectives

- Introduce you to larger programs and programs with multiple source files.
- Accustom you to reading someone else's code.
- Empower you with Makefiles.
- Introduce you to literature in computer science.
- Implement a party favor.

Recommended Reading

- Page 17 of http://www.howstuffworks.com/c.htm (http://www.howstuffworks.com/c.htm).
- Chapters 13, 15, and 18 of *Programming in C*.

diff pset3 hacker3

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

Academic Honesty

This course’s philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course’s material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy’s other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course’s final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course’s heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter to the Administrative Board and the outcome is Admonish, Probation, Requirement to Withdraw, or Recommendation to Dismiss, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course’s material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at Office Hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.

- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code on CS50 Discuss or elsewhere so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution in CS50 Vault to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate your solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Redeeming or attempting to redeem someone else's code for a late day.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.

- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Scores

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 SAT/UNS or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a satisfactory grade unless granted an exception in writing by the course's heads.

Shorts

- Head to https://www.cs50.net/shorts/3 (https://www.cs50.net/shorts/3) and watch the shorts on bubble sort, insertion sort, and selection sort. Then head to https://www.cs50.net/shorts/4 (https://www.cs50.net/shorts/4) and watch the short on `gdb`. (Phew, so many shorts! And so many sorts! Ha.) Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!

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

Getting Started

- 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

  ```
  update50
  ```

  to make sure your appliance is up-to-date. Then execute

  ```
  cd ~/Dropbox
  ```

  followed by

  ```
  wget http://cdn.cs50.net/2013/fall/psets/3/hacker3/hacker3.zip
  ```

  to download a ZIP of this problem set's distro into your appliance (with a command-line program called `wget`). You should see a bunch of output followed by:

  ```
  'hacker3.zip' saved
  ```

  If you instead see

  ```
  unable to resolve host address
  ```

your appliance probably doesn't have Internet access (even if your laptop does), in which case you can try running `connect50` or even restarting your appliance via **Menu > Log Off**, after which you can try `wget` again.

Ultimately, confirm that you've indeed downloaded `hacker3.zip` by executing:

```
ls
```

Then, run

```
unzip hacker3.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `hacker3` as well. Proceed to execute

```
cd hacker3
```

followed by

```
ls
```

and you should see that the directory contains two "subdirectories":

```
fifteen  find
```

Fun times ahead!

Searching

By default, this problem set is due by Thu 10/3 at noon. However, if you

- **complete this section before Wed 10/2 at noon (i.e., early),**
- **generate a coupon code confirming as much before Wed 10/2 at noon, and**
- **submit that coupon code at [https://www.cs50.net/coupons/3 (https://www.cs50.net/coupons/3)](https://www.cs50.net/coupons/3) before Wed 10/2 at noon,**

your deadline for the rest of the problem set's problems will be extended from Thu 10/3 at noon until Fri 10/4 at noon. If you do not submit a coupon code before Wed 10/2 at noon, this problem set, including this section, is still due by Thu 10/3. In other words, by tackling

part of the problem set early, you can extend your deadline for the rest!

- 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 actually generate truly random) numbers, one per line. (Cf. https://www.cs50.net/resources/cppreference.com/stdother/rand.html (https://www.cs50.net/resources/cppreference.com/stdother/rand.html).) 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`. (Cf. https://www.cs50.net/resources/cppreference.com/stdother/srand.html (https://www.cs50.net/resources/cppreference.com/stdother/srand.html).) 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. 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. (Know that an `unsigned int` is just an `int` that cannot be negative.) And 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 `clang` 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
    `clang` -ggdb -std=c99 -Wall -Werror -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 `clang` 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.

```
clang -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
    clang -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. At least, it should, but it won't find anything yet! That's where you come in. More on your role in a bit.

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 1,000 pseudorandom numbers to `find`, which then searches those values for `42`.

```
./generate 1000 | ./find 42
```

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 1000 > numbers.txt
```

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

```
./find 42 < 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 `core` (more on that soon!), `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 `search`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! (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 are essentially utility functions that might later prove useful to other programs as well, much like those in the CS50 Library.) Take a peek at `helpers.c` with `gedit`, and you'll see that `search` always returns `false`, whether or not `value` is in `values`. Re-write `search` in such a way that it uses linear search, returning `true` if `value` is in `values` and `false` if `value` is not in `values`. Take care to return `false` right away if `n` isn't even positive.

  When ready to check the correctness of your program, try running the command below.

```
./generate 1000 50 | ./find 2008
```

Because one of the numbers outputted by `generate`, when seeded with `50`, is `2008`, your code should find that "needle"! By contrast, try running the command below as well.

```
./generate 1000 50 | ./find 2013
```

Because `2013` is not among the numbers outputted by `generate`, when seeded with `50`, your code shouldn't find that needle. Best to try some other tests as well, as by running `generate` with some seed, taking a look at its output, then piping that same output to `find`, looking for a "needle" you know to be among the "hay".

Incidentally, note that `main` in `find.c` is written in such a way that `find` returns `0` if the needle is found, else it returns `1`. You can check the so-called "exit code" with which `main` returns by executing

```
echo $?
```

after running some other command. For instance, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2008
echo $?
```

you should see `0`, since `2008` is, again, among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `true`, in which case `main` (written by us) should return (i.e., exit with) `0`. By contrast, assuming your implementation of `search` is correct, if you run

```
./generate 1000 50 | ./find 2013
echo $?
```

you should see `1`, since `2013` is, again, not among the 1,000 numbers outputted by `generate` when seeded with `50`, and so `search` (written by you) should return `false`, in which case `main` (written by us) should return (i.e., exit with) `1`. Make sense?

When ready to check the correctness of your program officially with `check50`, you may execute the below. Be sure to run the command inside of `~/Dropbox/hacker3/find`.

```
check50 2013.hacker3.find helpers.c
```

Incidentally, be sure not to get into the habit of testing your code with `check50` before testing it yourself. (And definitely don't get into an even worse habit of only testing your code with `check50`!) Suffice it to say `check50` doesn't exist in the real world, so running your code with your own sample inputs, comparing actual output against expected output, is the best habit to get into sooner rather than later. Truly, don't do yourself a long-term disservice!

Anyhow, 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
```

- Need help? Head to cs50.net/discuss (https://www.cs50.net/discuss) or join classmates at office hours (https://www.cs50.net/ohs)!

- Assuming you got `find` to work before Wed 10/2 at noon and `check50` outputted only green smileys, here's how to receive a late day. No need to follow this bullet's steps otherwise.

  Go ahead and open a terminal window via **Menu > Programming > Terminal**. A big, black window should open (bigger than the one embedded in `gedit`). Navigate your way to `~/Dropbox/hacker3`, as with

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

  and then re-run `check50` as follows to output a "coupon code":

  ```
  check50 -c 2013.hacker3.find helpers.c
  ```

  Go ahead and highlight the code that you see, select **Edit > Copy**, then visit https://www.cs50.net/coupons/3 (https://www.cs50.net/coupons/3) using Chrome inside of the appliance, paste the code where prompted, as with control-v on your keyboard, or type it manually, and then click **Redeem**. You should be rewarded with an extension of 24 hours.

  You may (re-)submit coupons as many times as you'd like, but you'll be rewarded with no more than one 24-hour extension per problem set.

Sorting

- Alright, linear search is pretty meh. Recall from Week 0 and Week 3 that we can do better, but first we'd best sort that hay.

- Notice that `find.c` calls `sort`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully too 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.

  Now, 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[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 $O(n)$, where n is the array's size. Yes, this running time is possible because you may assume that each of the array's numbers wil be non-negative and less than `LIMIT`, a constant defined in `generate.c`. Leverage that assumption! However, realize that the array might contain duplicates.

  A previous version of this specification accidentally asked for a running time of $O(n^2)$ instead of $O(n)$, so we will accept either. But you are encouraged to (re-)try to achieve $O(n)$, since that's the intended challenge!

  Now, 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 finitely many 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.

  Anyhow, 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.

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 best to test your implementation of `sort`. But don't forget that `printf` and, per Week 3'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!

Incidentally, check out **Resources** on the course's website for a 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.

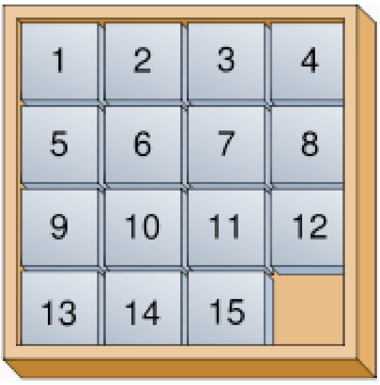
```
~cs50/hacker3/find
```

No `check50` for this one!

- Now that `sort` (presumably) works, it's time to improve upon `search`, the other function that lives in `helpers.c`. Recall that your first version implemented linear search. Rip out the lines that you wrote earlier (sniff) and re-implement `search` as Binary Search, that divide-and-conquer strategy that we employed in Week 0 and again in Week 3. 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. When it comes time to submit this problem set, it suffices to submit this new-and-improved version of `search`; you needn't submit your original version that used linear search.

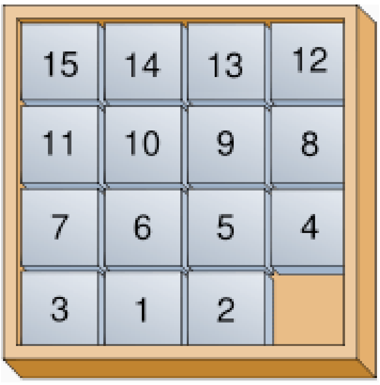
The Game Begins

- 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.



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. The puzzle is solvable from this configuration.



- Navigate your way to `~/Dropbox/hacker3/fifteen/`, and take a look at `fifteen.c` with `gedit`. (Remember how?) Within this file is an entire framework for the Game of Fifteen. The challenge up next is to complete this game's implementation.

  Implement God Mode for this game.

  First implement `init` in such a way that the board is initialized to a pseudorandom but solvable configuration. **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.** 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 dimension, 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 $d \leq 4$; you need not worry about 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 http://isthe.com/chongo/tech/comp/ansi_escapes.html (http://isthe.com/chongo/tech/comp/ansi_escapes.html) for more tricks.

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. In particular, `main` must only return `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. And be sure not to alter the staff’s implementation `save` or `main`'s usage thereof. 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` in the appliance, 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 (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 (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 $d \geq 4$), though, you might want to take a look at iterative deepening A* (IDA*) instead:

http://webdocs.cs.ualberta.ca/~tony/RecentPapers/pami94.pdf (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 (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 do click away if you stumble upon actual implementations (whether in C or other languages).

Alright, get to it, implement this game!

How to Submit

Step 1 of 2

- When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `hacker3` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `hacker3`, including any subdirectories (or even subsubdirectories!).

  ```
  zip -r hacker3.zip hacker3
  ```

  If you type `ls` thereafter, you should see that you have a new file called `hacker3.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm hacker3.zip`, then create a new ZIP file as before.)

- Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.net/submit (https://www.cs50.net/submit), logging in if prompted.

- Click **Submit** toward the window's top-left corner.

- Under **pset3** on the screen that appears, click **Upload New Submission**.

- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.

- Navigate your way to `hacker3.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `hacker3.zip`, click it once to select it, then click **Open**.

- Click **Start upload** to upload your ZIP file to CS50's servers.

- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.net/submit (https://www.cs50.net/submit) and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://forms.cs50.net/2013/fall/psets/3/ (https://forms.cs50.net/2013/fall/psets/3/) where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 3.

Last updated 2013-10-30 17:27:17 EDT