Table of Contents

- tl;dr
- Distribution
  - Downloading
  - Understanding
- Specification
  - `search`
  - `sort`
- Walkthroughs
  - search
  - sort
- Usage
- Testing
  - `check50`
- Staff’s Solution
- Hints
- FAQs
- Changelog

Find

tl;dr

Implement a program that finds a number among numbers, per the below.

```
$ ./generate 1000 | ./find 42
Didn't find needle in haystack.
```

Distribution

Downloading

```
$ wget https://github.com/cs50/problems/archive/find.zip (https://github.com/cs50/p
$ unzip find.zip
$ rm find.zip
$ mv problems-find find
$ cd find
$ ls
Makefile    find.c      generate.c  helpers.c   helpers.h
```

Understanding

Implemented in `generate.c` is a program that uses a "pseudorandom-number generator" (via a function called `drand48`) to generate a whole bunch of random (well, pseudorandom, since computers can't actually generate truly random) numbers, one per line, each of which is in [0, `LIMIT`), where `LIMIT` is a constant defined within the file, so to speak. That is, each is greater than or equal to 0 and less than `LIMIT`.

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 square brackets 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 `drand48` by first calling `srand48` (another function whose purpose is to "seed" `drand48`) with an argument of, say, `0`, and then call `drand48` itself three times, `drand48` might return `0.170828`, then `0.749902`, then `0.096372`. But if you instead seed `drand48` by first calling `srand48` with an argument of, say, `1`, and then call `drand48` itself three times, `drand48` might instead return `0.041630`, then `0.454492`, then `0.834817`. But if you re-seed `drand48` by calling `srand48` again with an argument of `0`, the next three times you call `drand48`, you'll again get `0.170828`, then `0.749902`, then `0.096372`! 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. (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 `drand48` and `srand48`, recall that you can execute:

```
man drand48
```

and:

```
man srand48
```

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. 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:
	clang -ggdb3 -O0 -std=c11 -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. 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, 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`. 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 -ggdb3 -O0 -std=c11 -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 -ggdb3 -O0 -std=c11 -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 `get_int` 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.

It 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?)

Notice now that, in `find.c`, `main` calls `search`, a function declared in `helpers.h`. Unfortunately, we forgot to implement that function fully in `helpers.c`! Indeed, take a peek at `helpers.c`, and you'll see that `search` always returns `false`, whether or not `value` is in `values`. 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.

Notice too, per `helpers.h`, that the prototype for `search` is:

```
bool search(int value, int values[], int n);
```

And the prototype for `sort` is:

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

Both functions take an array, `values`, as one of their arguments as well as an integer, `n`, the size of that array. That's because, when passing an array to a function, you have to pass in its size separately; you can't infer an array's size from the array itself.

Specification

Complete the implementation of `find` by completing the implementation of `search` and `sort` in `helpers.c`.

`search`

- Your implementation must return `false` immediately if `n` is non-positive.
- Your implementation must return `true` if `value` is in `values` and `false` if `value` is not in `values`.
- The running time of your implementation must be in *O*(log *n*).
- You may not alter the function’s declaration. Its prototype must remain:

```
bool search(int value, int values[], int n);
```

`sort`

- Your implementation must sort, from smallest to largest, the array of numbers that it’s passed.
- Assume that each of the array’s numbers will be non-negative and less than 65,536. But the array might contain duplicates.
- The running time of your implementation must be in *O*(*n*), where *n* is the array’s size. Yes, linear! Keep in mind that 65,536 is a constant.
- You may not alter the function’s declaration. Its prototype must remain:

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

Walkthroughs

search

sort

Usage

Your program should behave per the examples below. Assumed that the underlined text is what some user has typed. (`^d` represents the ctrl-d character described above)

```
$ ./find 42
50
43
^d
Didn't find needle in haystack.

$ ./find 42
50
42
^d
Found needle in haystack!
```

Testing

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

```
./generate 1000 50 | ./find 127
```

Because one of the numbers outputted by `generate`, when seeded with `50`, is `127`, your code should find that "needle"! By contrast, try running the command below as well.

```
./generate 1000 50 | ./find 128
```

Because `128` 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 127
echo $?
```

you should see `0`, since `127` 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 128
echo $?
```

you should see `1`, since `128` 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?

`check50`

```
check50 cs50/2017/x/find/more
```

Staff’s Solution

```
~cs50/pset3/find
```

Hints

Before you implement `search` in *O*(log *n*) time, you might want to implement it temporarily in *O*(*n*) time, as with linear search, if only because it’s a bit easier to get right. That way, you can move on to `sort`, knowing that `search` already works. And once `sort` works, you can go back and re-implement `search` in *O*(log *n*) time, as with binary search. Just remember to!

Ultimately, you are welcome to implement `search` iteratively (with a loop) or recursively (wherein a function calls itself). 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.

We leave it to you to determine how best to test your implementation of `search` and `sort`. But don’t forget that `eprintf` is your friend while debugging! And don’t forget that you can generate the same sequence of pseudorandom numbers again and again by explicitly specifying `generate`'s seed.

FAQs

None so far! Reload this page periodically to check if any arise!

Changelog

- 2017-03-09
  - Clarified usage
- 2016-09-21
  - Update to `search` walkthrough
- 2016-09-17
  - Corrected "non-negative" to "non-positive."
  - Clarified that the prototype of `search` cannot be alterered either.
  - Removed incorrect mention of $O(n^2)$.
- 2016-09-16
  - Initial release.