Lecture notes: January 11, 2017

Topics:

1. GitHub (live demo)
    I. Cloning a git repo
        a. Using HTTPS on ECN machines
        b. Using SSH (including generating SSH keypair)
    II. How to add and commit
    III. How to push
    IV. How to tag
    V. How to pull (if you're developing elsewhere)
2. Makefiles
    I. Format
    II. Macros
3. Selection sort

GitHub

We did a live demo showing the following steps
1. Show HTTPS clone on ecegrid machine
2. Show SSH setup and clone on ecegrid machine
3. Show git status, git add, git commit, git push
4. Show git tag
5. Show git pull on laptop

Makefiles

Makefiles let you define complicated sets of commands to build your projects.

Makefiles consist of a series of rules:

```
[target] : [dependences]
        [command 1]
        [command 2]
        …
```

A rule *target* is the name of the rule. The *dependences* are the files the rule depends on. The *commands* are what to do when the rule is "fired." Note: there must be a tab before each command.

A rule is fired in one of two ways: (i) it is directly invoked (by calling "make [target]") or (ii) it is invoked by another rule that is fired.

When a rule is fired, it goes through the following process:
1. If a *dependence* has a rule in the Makefile, fire that rule (using this same process)

2. Once all dependences have been fired, check to see if *target* is "out of date": interpret *target* as a filename, and see if the timestamp on the file is older than the time stamp on any of its dependences. If it is, the target is "out of date." If there is no file named *target*, the target is always assumed to be out of date. If there are no dependences and *target* exists, target is assumed to be up to date.
3. If the target is out of date, execute the list of commands

You can use Makefiles to orchestrate complicated build processes.

If you type "make" without a target, make will fire the first rule in the Makefile.

We usually define a target called "clean" whose job it is to clean up any intermediate files generated during the build process. This can also be used to remove all generated targets to force recompiling everything.

Makefiles also let you define macros to reuse the same commands over and over. For example, we can define GCC as a macro that invokes gcc the way we want:

```
DEBUG = -DDEBUG
CFLAGS = CFLAGS = -std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror
GCC = gcc $(CFLAGS) $(DEBUG)
```

Note that we use $(MACRO_NAME) to insert the macro into other places, including commands.

Makefiles can get much more complicated than this, but their full power is beyond the scope of this course.

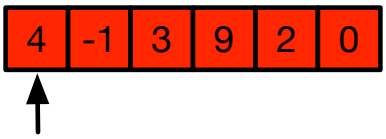
Selection Sort

Readings: Forouzan and Gilberg, pp 491–493

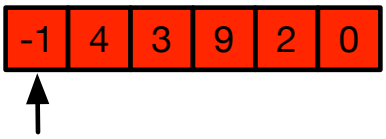
Selection sort is one particular sorting algorithm that sorts an array using the following procedure:

Divide the array up into two pieces (we'll call them "sorted" and "rest"). *Sorted* is the portion of the array that is *already sorted*. *Rest* is the rest of the array. One thing that is always true (an *invariant*) is that all the elements in *sorted* are smaller than any of the elements in *rest*.)

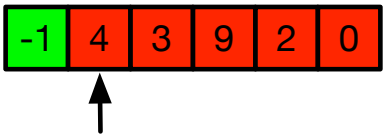
Selection sort works by slowly growing the sorted side of the array and shrinking the rest of the array. Think of this as having a cursor. When the sorting starts, we don't know if *any* of the array is sorted, so our cursor starts out pointing to the first element of the array. Everything to the left of the cursor (colored green, in this case nothing!) is sorted:



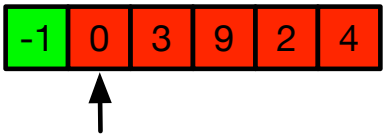
Everything from the cursor to the right (colored red) is the *rest* of the array. We then scan through the *rest* of the array to find the smallest element, and swap it with the element at the cursor (this might be the element itself!):



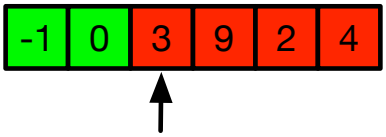
Because we just moved the smallest element to the cursor, we can now move the cursor up one:



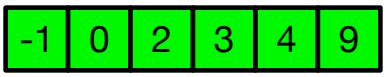
Note that our rules still hold: everything to the left of the cursor is sorted, and everything from the cursor right is larger than anything in the sorted part of the array. Now we repeat the process, finding the smallest element in the *rest* of the array and swapping it with the cursor:



And we can now move the cursor to the right again, restoring our properties. The *sorted* part of the list is sorted, and the *rest* of the list is larger than anything in the *sorted* part of the list:



Note that each time we repeat this process, the cursor moves one to the right. The *sorted* list gets longer, and the *rest* of the list gets shorter. In this manner, we eventually sort the list:



We will use *pseudocode* for most of our code examples in class. This lets us quickly explain the structure of an algorithm without worrying about nitty-gritty details of correct C syntax. It also means that we can describe an algorithm without giving you code that you can just copy for an assignment!

This is where we left off in class on 1/11. We will briefly discuss the pseudocode below on 1/13.

Here is some pseudocode for selection sort:

```
int input[N] = //input
cursor = 0 //initial position of the cursor
for (cursor = 0; cursor < N; cursor++)
      //sorted list from [0,cursor)
      //rest of the list from [cursor, N)
      for (i = cursor; i < N; i++)
            //search the rest of the list to find the smallest value
      //swap the smallest value with the value at input[cursor]
```

Note that the outer loop (highlighted in green) is doing the job of moving the cursor over. We eventually want to move it all the way to the end of the array, so we're going from 0 up to N (once cursor = N, it's past the end of the array—remember, an N element array has elements from 0 to N-1). The inner loop (highlighted in red) is doing the job of searching the *rest* of the array, which starts at cursor.