Lecture notes: Week 1

In Week 1, we covered:

1. The syllabus
2. The Phone Dropping Problem
3. Git and GitHub
4. Selection Sort

Phone Dropping Problem

Your goal is to figure out from how many stories high you can drop a phone from before it breaks. If you have one phone, the strategy is to start by dropping the phone from the 1st floor and work your way up floor by floor until the phone breaks. What is the optimal strategy (minimizes the worst case scenario of number of drops) if you have two phones? Assume you have a 100 story building.

Potential Approaches

Binary search (start at floor 50, then try floor 25 or 75, and continue, cutting it in half each time).
Issues: Usually, binary search is a great tool for these sorts of problems. The issue here is that you only have two phones, and once the first phone breaks you're stuck with just one phone (and hence have to use the one-phone solution of trying every floor). So if the first phone breaks when you try floor 50, you have to use the second phone to try every floor between 1 and 49 (worst case scenario: 50 drops total)

Intervals (Divide the building into equal intervals. Use the first phone to test the "top" of each interval until it breaks, use the second phone to test the floors in that interval).
Issues: This is a good general strategy.

A good guess might be to divide the building up into 10 even intervals. The first phone tests floors 10, 20, 30, etc., and if it breaks (say at floor 40), use the second phone to test floors 31, 32, etc. The worst case scenario here is that you need 19 drops (the first phone breaks on floor 100—10 drops—then you have to try 9 additional drops from floors 91 to 99).

In general, the worst case scenario is going to be (*number_of_floors*/*interval_size* + (*interval_size* - 1)).

Variable size intervals But we can do better. Note that if phone A breaks in the first interval, you only have to do 9 more drops with phone B for a total of 10 drops. If phone A breaks in the third interval, you still have to do 9 more drops with phone B for a total of 12 drops. Getting up to higher intervals means "spending" more drops with phone A before you use your 9 drops with phone B. If the final answer is in a higher interval, it will take you more drops to find it.

But what if we make higher intervals smaller? That way we “spend” drops getting to the *k*th interval, and then use fewer drops exploring that interval: If the final answer is in the *k*th interval, the number of drops needed to explore that interval is: *k* + size of *k*th interval.

So to minimize the worst case scenario, we want *k* + size of *k*th interval to be constant for all *k*.

That means we want the size of interval 1 to be 1 bigger than the size of interval 2, which should be 1 bigger than the size of interval 3, and so on.

Flipping things around, we want the *last* interval to only have 1 story in it, and the *second-to-last* interval to only have 2 stories in it, and so on.

All that is left is to figure out how many intervals we need to cover all 100 stories!

How many stories do we cover if we have *n* intervals? We can turn to some simple math:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

(Later in class, we’ll use a technique called *induction* to prove that this is true)

So if we want to cover 100 stories, we need:

$$\frac{n(n+1)}{2} = 100$$

Or *n* = 14 (technically, n = 13.65 or so, but we can’t have intervals that are fractional stories).

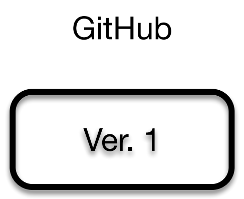
So we can create 14 intervals, with the top interval being 1 story, the second being 2 stories, etc. and then guarantee that we will need no more than 14 drops to figure out how high we can drop a phone from before it breaks.

Git and GitHub

We covered how to set up SSH keys on GitHub (see notes on Piazza and links in hw0 README for more details). We also covered how the git commands `add`, `commit`, `push`, and `pull` worked.

What is a Git repository?

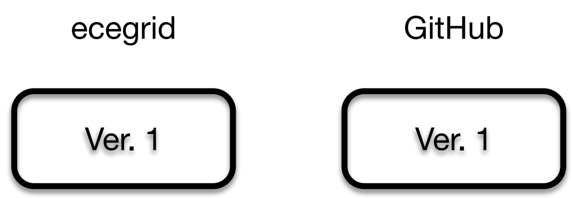
A git repository is a collection of *versions* of a bunch of files. When you set up your initial git repository (e.g., by clicking on a homework link), you have an initial repository on GitHub that has a single version of code:



That version of code has a unique identifier associated with it. It's a long string of letters and numbers that is unique *to that version of the code*. We'll call this "Ver. 1." No one else's repository contains the same unique identifier, unless you explicitly copy that code somewhere else. So let's do that:

```
> git@github.com:PurdueECE264/hw0-milindkulkarni.git
```

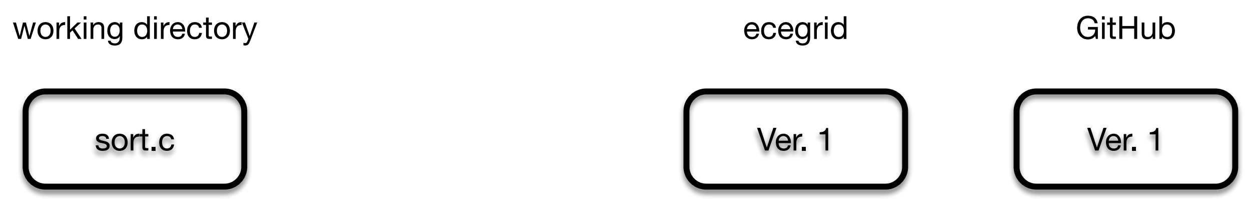
So what does this do? It copies *the entire repository from GitHub to your local machine*. So if we were doing this on ecegrid, we'd now have this situation:



Version 1 now exists in two places: on GitHub, and on your local machine. These are exact copies of each other, and they have the same unique identifier.

Creating new versions

Now, suppose we start editing a file on ecegrid, perhaps `sort.c`. This *doesn't change version 1* on ecegrid. Versions that git knows about *do not change* (unless you do some extra git magic). Instead, git knows that the changes you've made to `sort.c` make it different than version 1, and your *working directory* has changes:

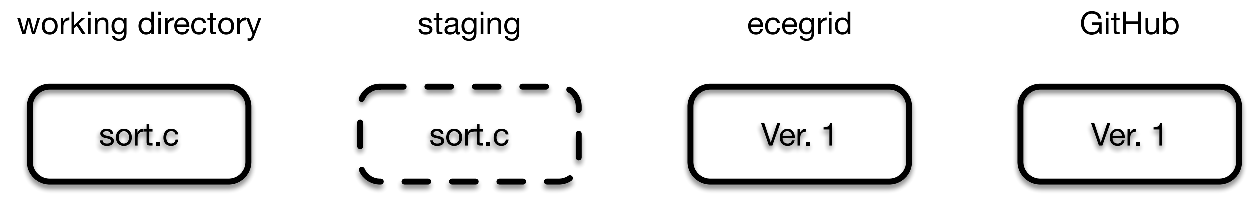


You can edit files to your heart's content on your local machine, and these changes will only be in your working directory (these are also what you'll see if you open up a file browser and look at the files; git's versions are hidden).

Suppose you get `sort.c` to the point where you are happy with it, and you want to officially make a new version that git knows about. To do this, you add the file:

```
> git add sort.c
```

This copies sort.c into a *staging area*. Think of this as a “prospective new version.”

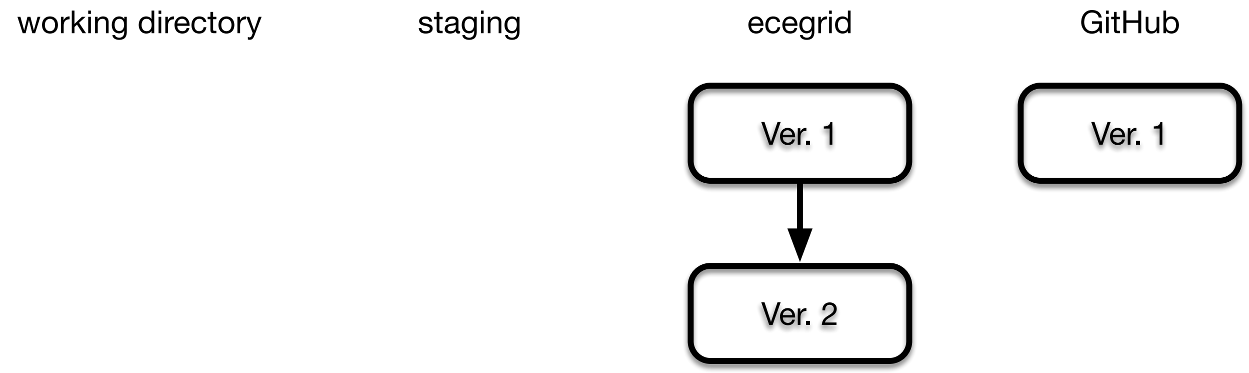


You can keep adding things to the staging area to build up what you want the new version to look like. The way to think about this is that the new version will be version 1, plus whatever changes are in the staging area. Once the new version looks the way you want, you can tell GitHub to create it:

```
> git commit -m “sample message”
```

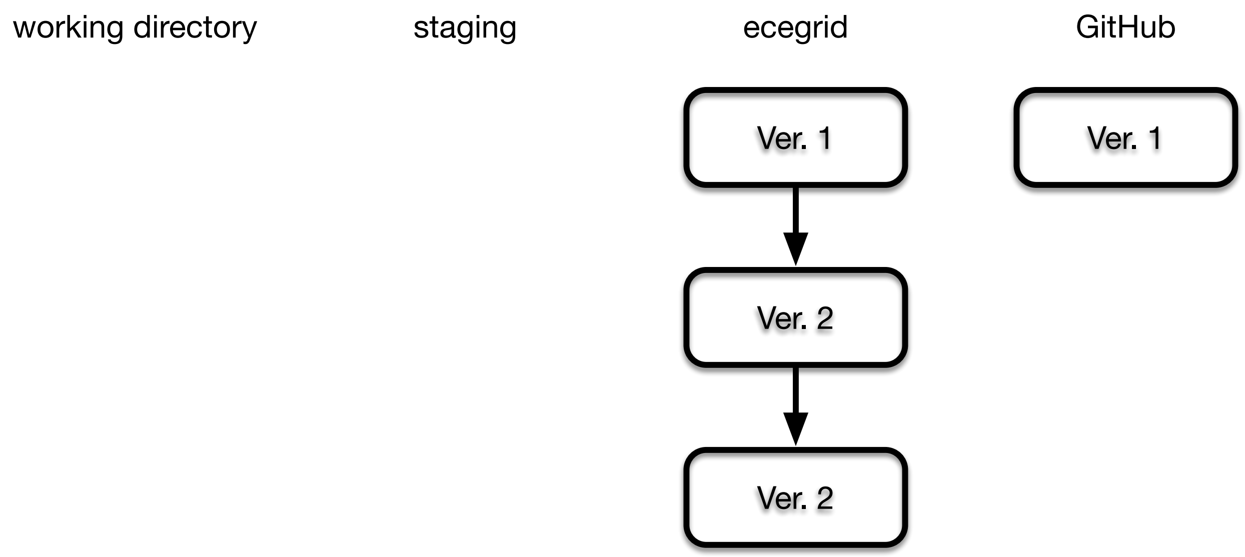
(You can make “sample message” whatever you want. Ideally, it would be some short text that describes what this new version changes, like “added working sort routine.” Just make sure that it’s in quotes.)

Once you’ve done this, here’s what you have:



There’s now a version 2! This version is still local to your machine. Note that there is now nothing in the working directory—it has exactly the same content as version 2—and nothing in the staging area—there’s no new version being prepared.

You can go through this whole process again, to create *another* version:



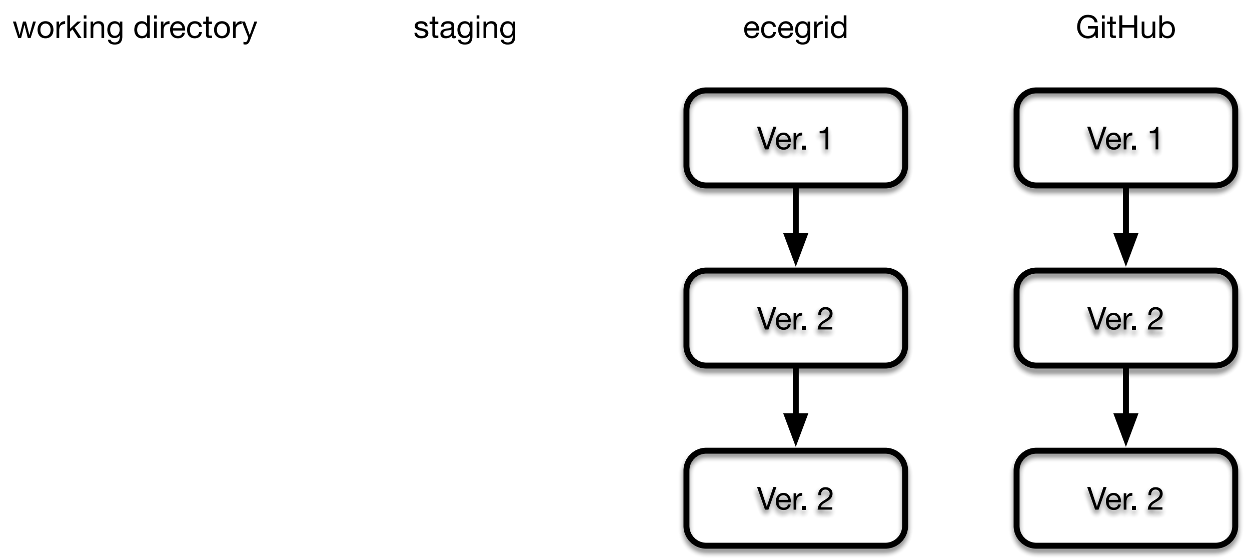
And so on. Note that all of these versions are just on ecegrid. From GitHub's perspective, your repository still only has version 1.

Synchronizing with GitHub

To synchronize your repositories, you can *push* the versions from ecegrid back to GitHub:

```
> git push
```

Which gives you this:



Tagging (see the README for details) is a way of giving a name to a specific version of code. That way you don't have to refer to it by the long string of letters and numbers. By default, when you run git tag, it names the latest version of code in your repository. When we grade your code, we will look for a version named `final_ver`, and grade exactly that code, regardless of whatever else is in your repository.

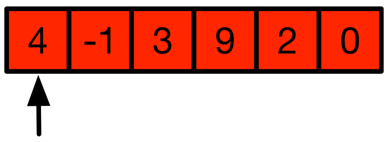
Don't forget to push the tags to GitHub so we can see them! If your code is just on ecegrid, we won't know it's there. You can check that your tag made it to GitHub (and that it's tagging the right version of the code) by clicking the "tag" link right above your code on GitHub.com

Selection Sort

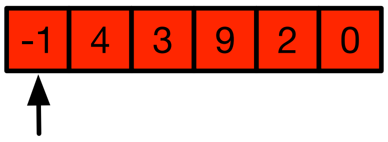
Selection sort is one particular sorting algorithm that sorts an array using the following procedure:

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*.)

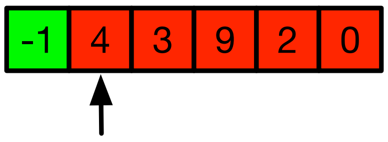
Selection sort works by slowly growing the sorted side of the array and shrinking the rest of the array. Think of this as having a cursor. When the sorting starts, we don't know if *any* of the array is sorted, so our cursor starts out pointing to the first element of the array. Everything to the left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



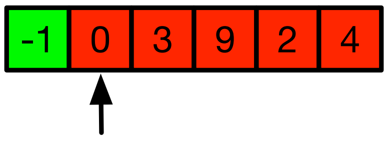
Everything from the cursor to the right (colored red) is the *rest* of the array. We then scan through the *rest* of the array to find the smallest element, and swap it with the element at the cursor (this might be the element itself!):



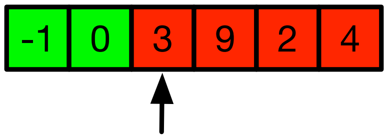
Because we just moved the smallest element to the cursor, we can now move the cursor up one:



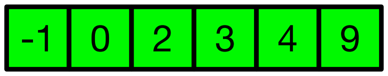
Note that our rules still hold: everything to the left of the cursor is sorted, and everything from the cursor right is larger than anything in the sorted part of the array. Now we repeat the process, finding the smallest element in the *rest* of the array and swapping it with the cursor:



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:



Selection Sort pseudocode

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!

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.

Selection Sort runtime

Some of you may have noticed that selection sort takes a very long time to run on large inputs. How long, exactly?

Let's count *iterations*: how many times does the inner loop (which searches for the minimum value in the *rest* of the array) run?

```
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]
```

The inner loop runs *N* times, and each time it runs, it runs for (*N* - *cursor*) iterations. Cursor takes on every value from 0 to *N - 1*:

$$\sum_{i=0}^{N-1} N - i$$

That summation is the same as:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Note, also, that most of the work happens in the inner loop, so how long selection sort takes is dominated by how long that inner loop takes. Trying to be precise about just how long the inner loop takes is tricky: depending on how you wrote it, it may take more or fewer instructions to execute. But what matters is, no matter how you wrote that inner loop, *if we make N twice as big, the inner loop will run about four times as many times*! That's the dominating factor here: double the input, take four times as long. So all that really matters is the quadratic term. The next +N or /2 doesn't really matter.

Thinking about run time this way is called *asymptotic analysis*, and we'll come back to it later in class.