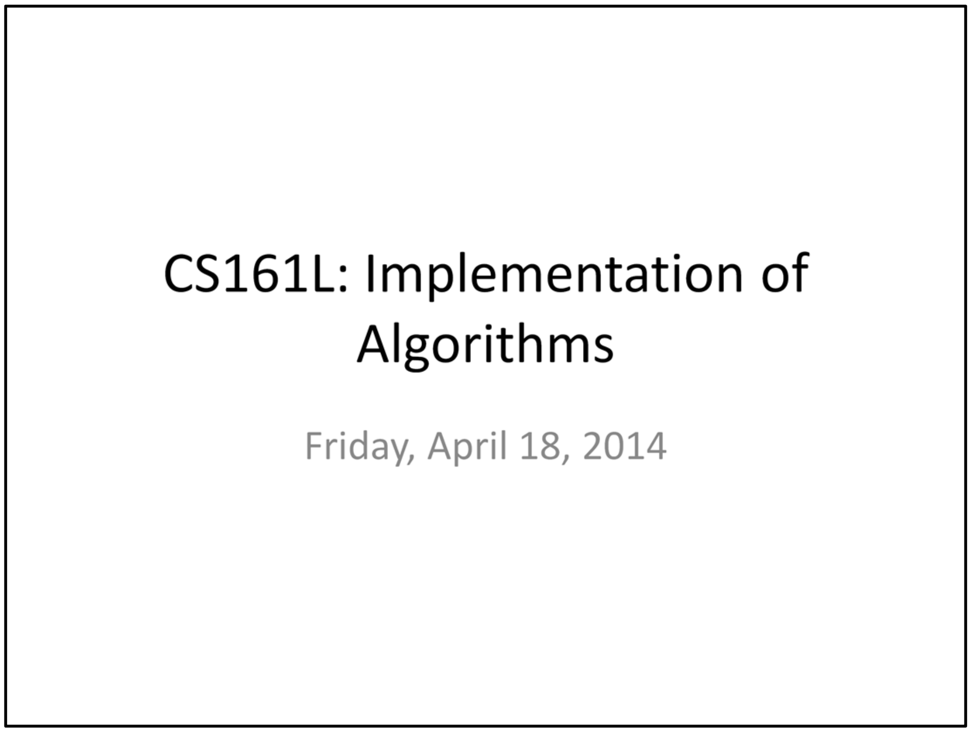
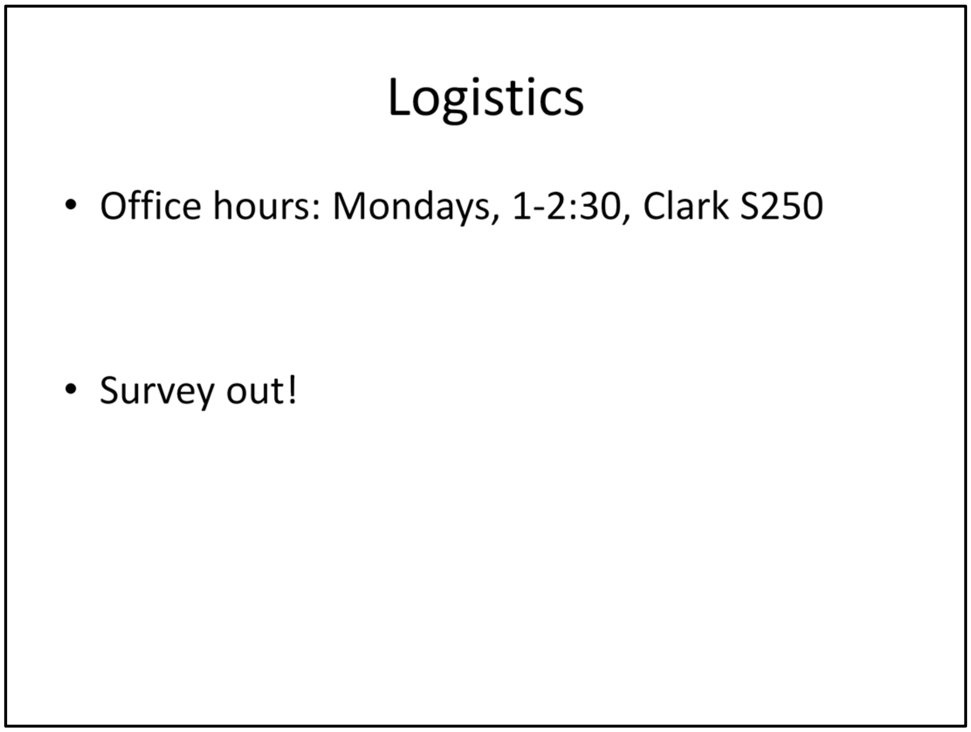


CS161L: Implementation of Algorithms

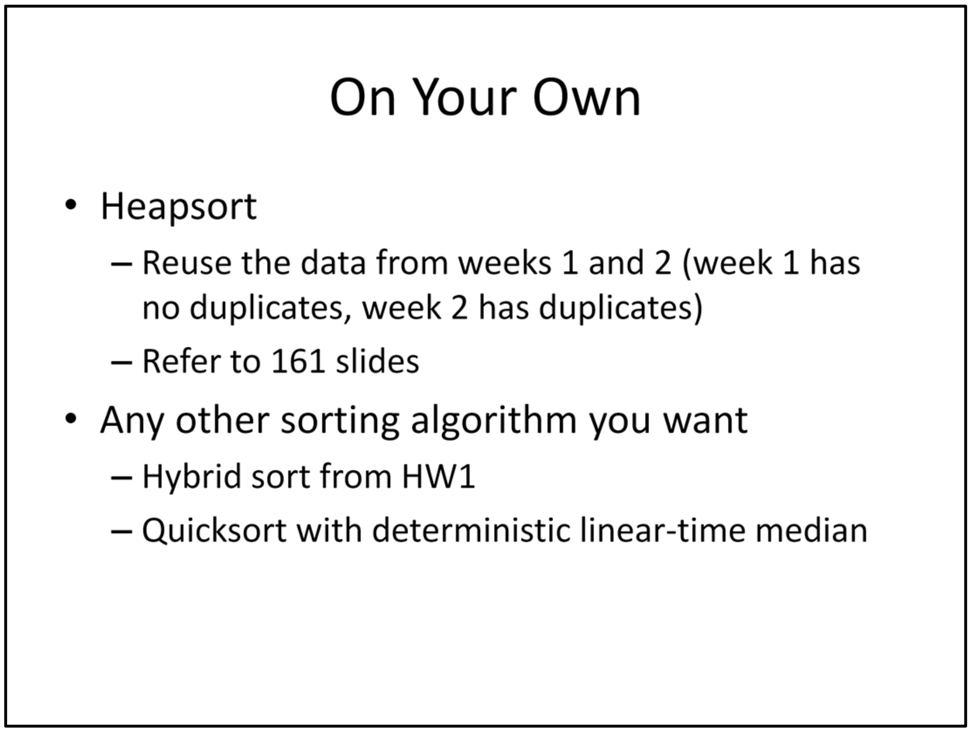
Friday, April 18, 2014



Logistics

- Office hours: Mondays, 1-2:30, Clark S250
- Survey out!

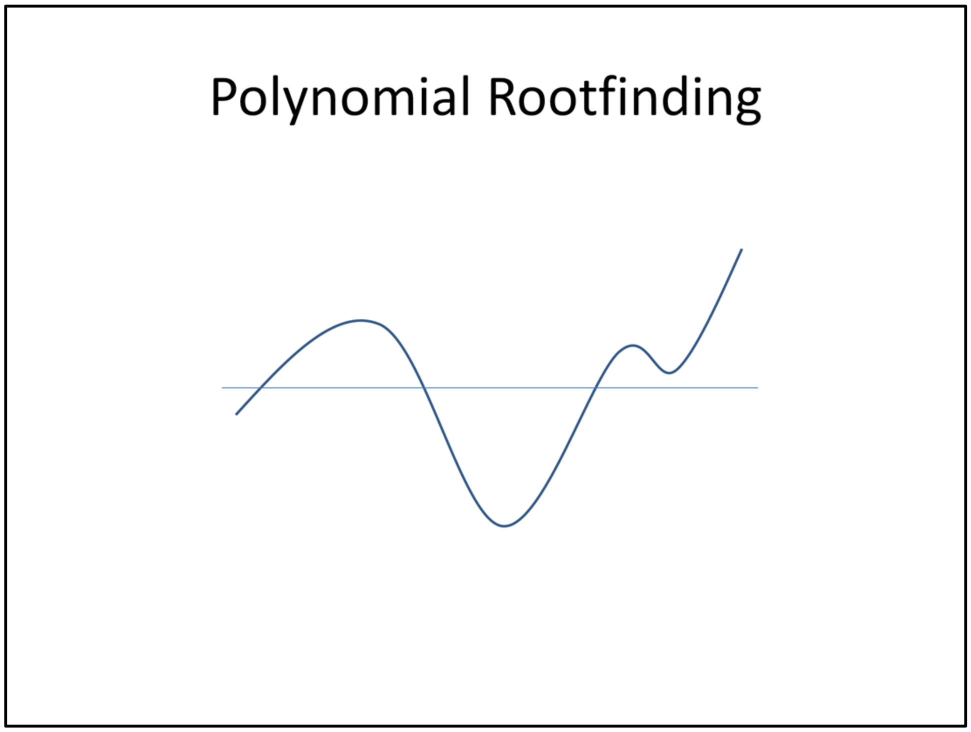
Before we start going over the coding problem for today, I'd like to make a couple announcements. First, I've noticed that rather a lot of submissions don't actually get turned in until pretty late Monday night, so to help with that, I'm going to hold some office hours on Monday afternoons if you have some bug you're stuck on. Second, I've posted a survey about the class so far on Piazza; when you get the chance, please fill it out so I can plan the rest of the course accordingly.



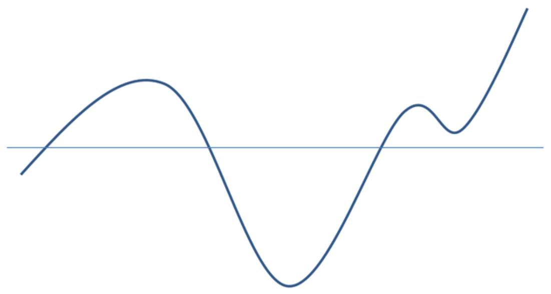
On Your Own

- Heapsort
  - Reuse the data from weeks 1 and 2 (week 1 has no duplicates, week 2 has duplicates)
  - Refer to 161 slides
- Any other sorting algorithm you want
  - Hybrid sort from HW1
  - Quicksort with deterministic linear-time median

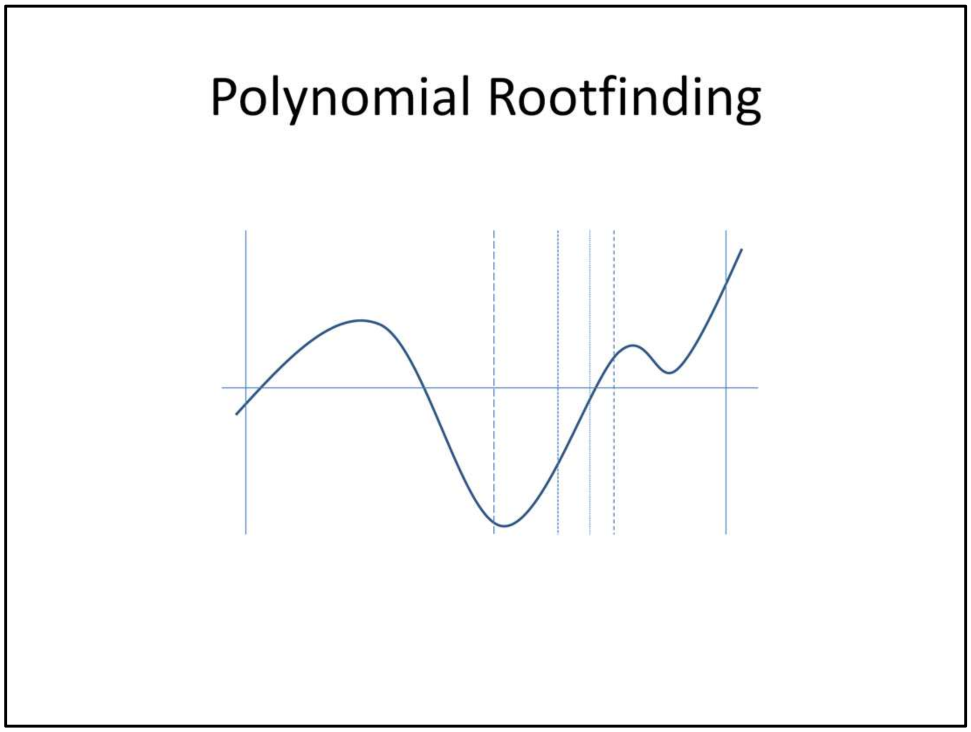
If you've checked the web site, you may have noticed that there's only one problem this week, and it's pretty different from the algorithms explicitly covered in 161 lecture. The reason for that is I'd like to give you folks more practice applying the general algorithm design techniques to some new problems. Also, we've now implemented 3 sorting algorithms, and I'm guessing that some of you may be sick of sorting by this point. That said, if you want to practice the algorithms from class, such as heapsort, you can do so with the data I've already given you. Heapsort in particular is a good one to try coding up.



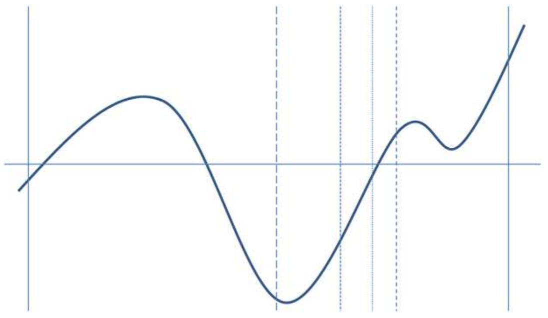
Polynomial Rootfinding



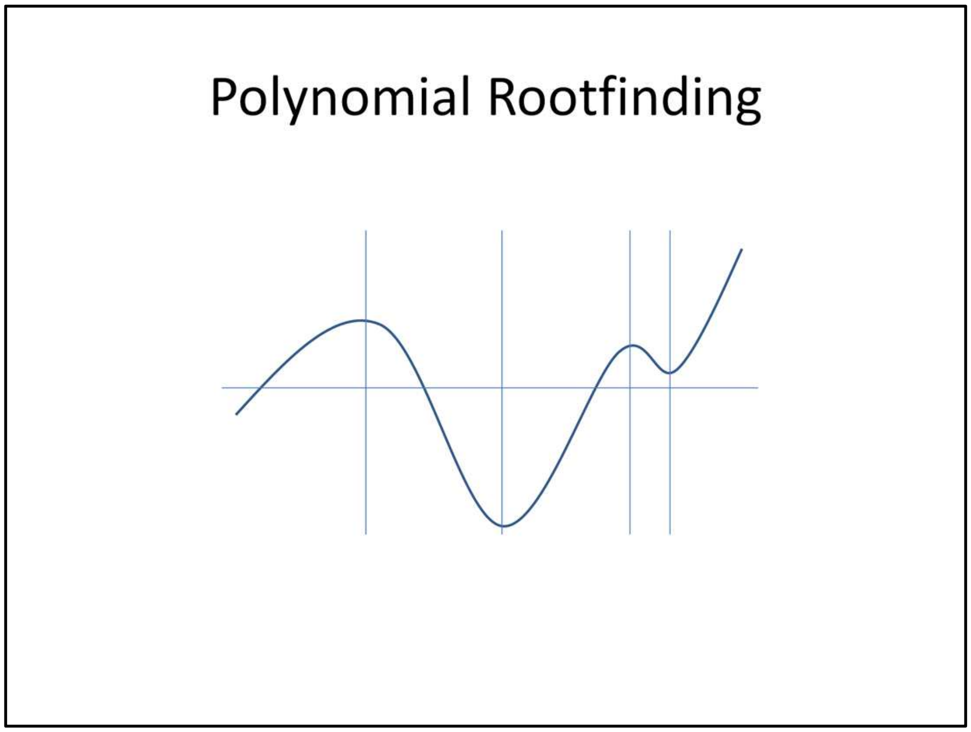
All right, now to discuss this week's problem, that of finding all the real roots of a polynomial. This problem is our first adventure into continuous problems, which can have some nasty edge cases, but for today we're going to ignore those edge cases and focus on the main idea. So how would we find the roots of a polynomial? Well, there's this theorem called the Intermediate Value Theorem which tells us if we have a continuous function, such as a polynomial, and we're negative here on the left and positive here on the right, or vice versa, we have to be zero somewhere in the middle. This makes sense intuitively; if we start below the x-axis and end up above it, we have to cross it at some point.



Polynomial Rootfinding

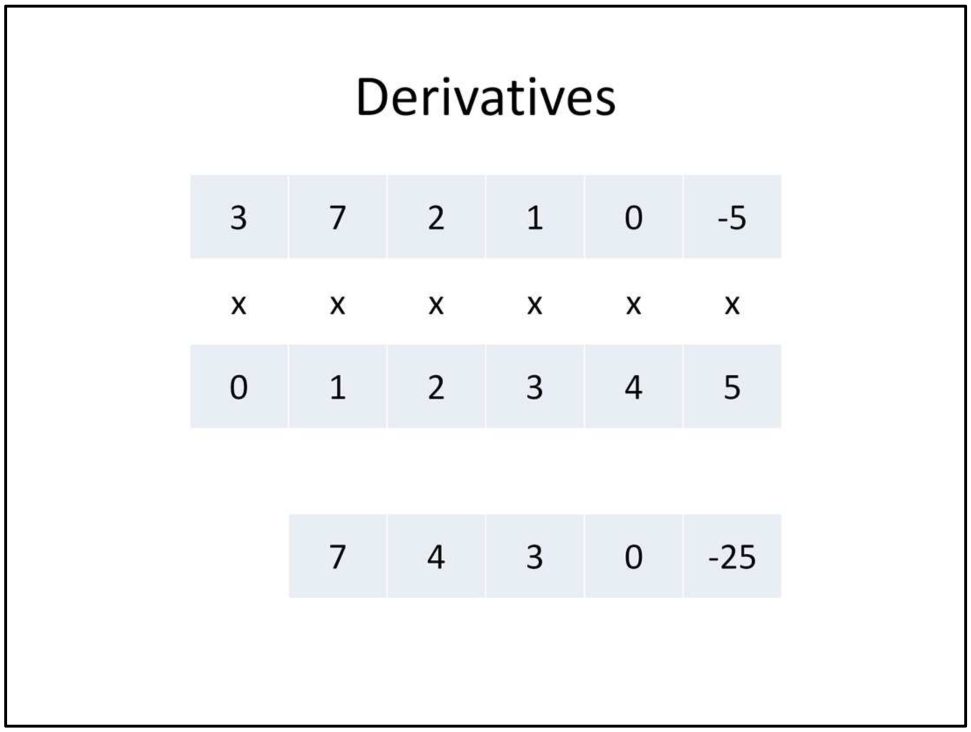


So how can we take advantage of this? Well, we could check the value right in the middle of our interval. If it's on the same side of the x-axis as our left point, then it's on opposite sides of our right point, so we can recurse into the right half of our interval. Otherwise, it's on opposite sides of our left point, so we can recurse into the left half. This looks a LOT like the binary search we apply to arrays. Unlike arrays, though, we're working on a continuous interval, so our stopping condition is different. In this case, we keep cutting the interval in half until it's within a certain tolerance; then any part of the interval (including, say, the left endpoint) is a suitable approximation of the root. This process is called bisection. Now, this will give us A root of the polynomial. But how do we find ALL the roots of the polynomial? You notice that there are 3 roots in this interval, yet we skipped over two of them.



Polynomial Rootfinding

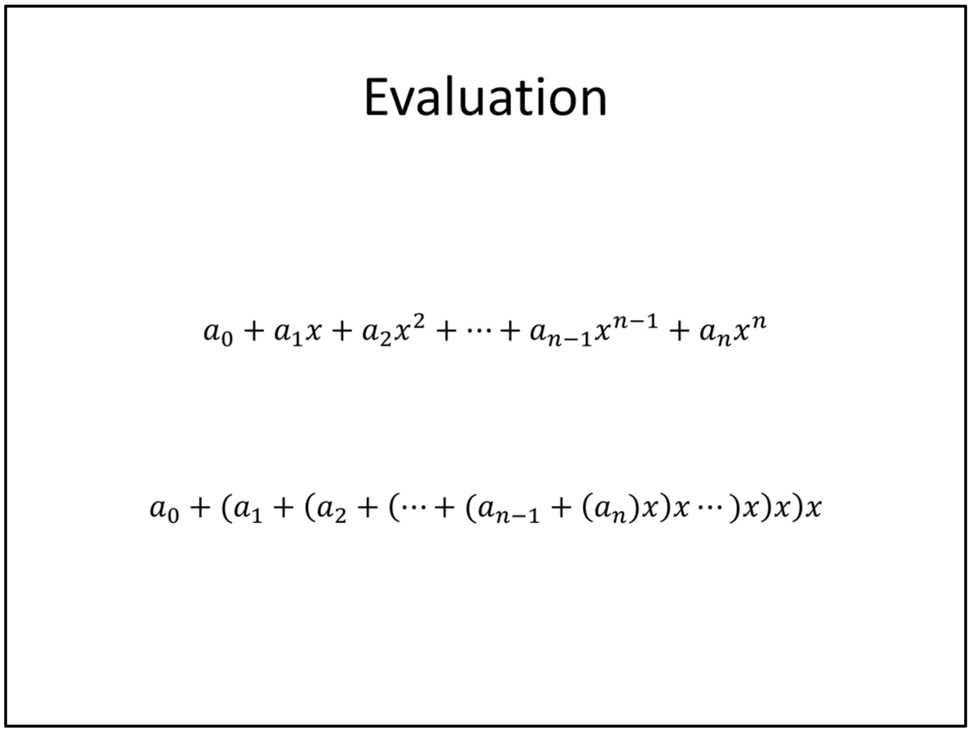
Well, one thing we can observe is that this problem comes from the fact that our function goes both up and down within our interval. If we could split our function into pieces that each go only up or down, we could run bisection on each piece and be confident that we didn't skip over any roots. But how do we do that? Well, the points at which we switch from going up to going down and vice versa, which we call critical points, have a slope of 0. This means that the points at which we want to split our function are the roots of the derivative of our function. Notice that sometimes we'll have pieces that don't have roots, and that's fine; we just need to make sure each piece has at MOST 1 root. Also notice that since our function is a polynomial, the derivative of our function is a polynomial of degree one less. That means we can get away with recursively finding the roots of our derivative, since we have to bottom out eventually. Keep in mind that there are TWO recursive algorithms going on here. There's the bisection algorithm, which we'll only feed intervals that have exactly one root. And then there's the recursive processing of critical values, which we use to FIND the intervals that we feed to the bisection algorithm.



Derivatives

3	7	2	1	0	-5
x	x	x	x	x	x
0	1	2	3	4	5
	7	4	3	0	-25

Now, for a little bit of math review. How do we take the derivative of a polynomial? Well, for each term, we take its degree, multiply it with its coefficient, and reduce the degree of the term by one. In our representation, this is pretty simple to do. We take each entry in our array and multiply it with its index in the array. Then we throw away the first entry, leaving an array of size one less. So here, the 7 is the entry at index 0 in our derivative coefficient array, and the -25 is at index 4.

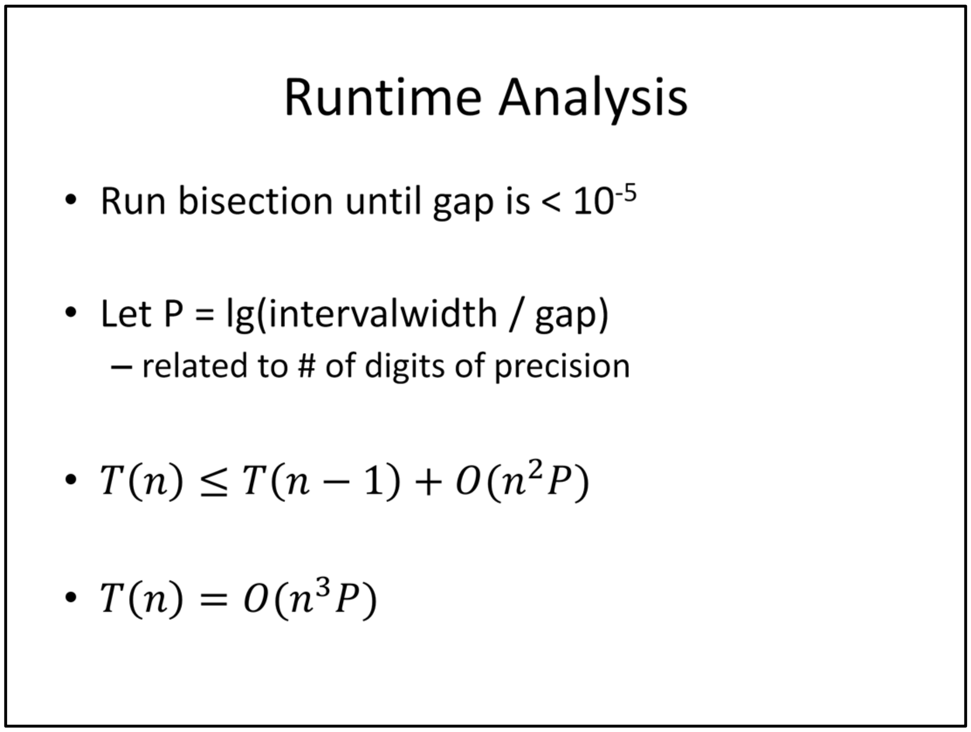


Evaluation

$$a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

$$a_0 + (a_1 + (a_2 + (\cdots + (a_{n-1} + (a_n)x)x \cdots)x)x)x$$

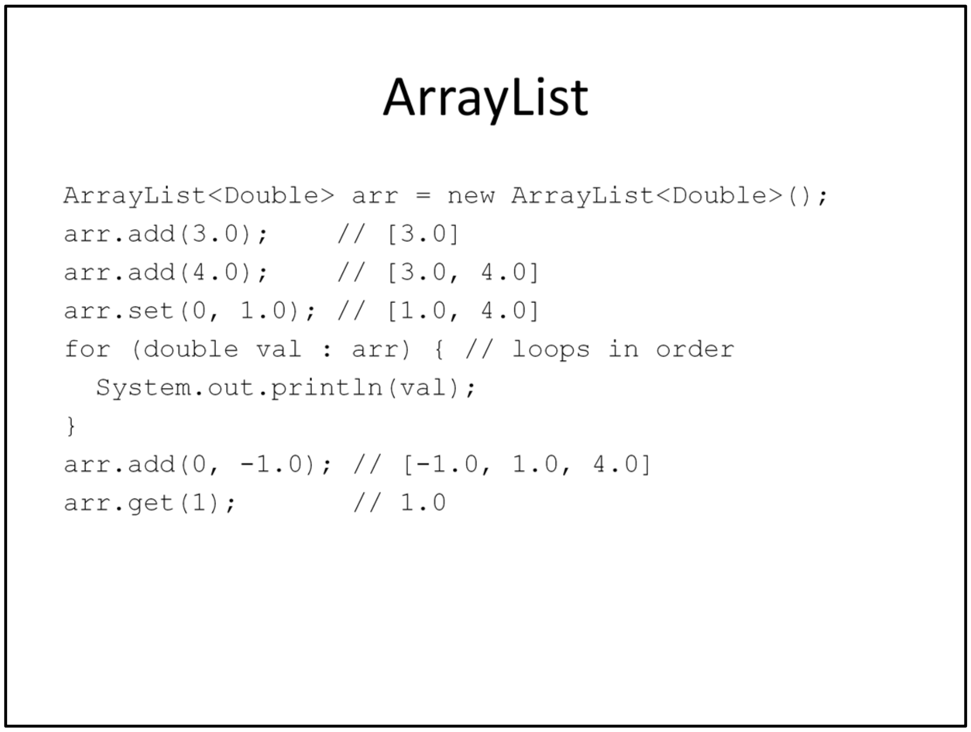
Next, how do we evaluate a polynomial at a given point? The short answer is "plug it in", but if we plug in values as written, we're going to have to use order n^2 multiplications and order n additions. If we do a little algebra, though, we can derive a formula that only takes n multiplications and n additions to evaluate. This is important since we have to evaluate a polynomial every time we want to split an interval, so we're going to be doing a lot of evaluations.



Runtime Analysis

- Run bisection until gap is < 10^{-5}
- Let P = lg(intervalwidth / gap)
  - related to # of digits of precision
- $T(n) \leq T(n-1) + O(n^2 P)$
- $T(n) = O(n^3 P)$

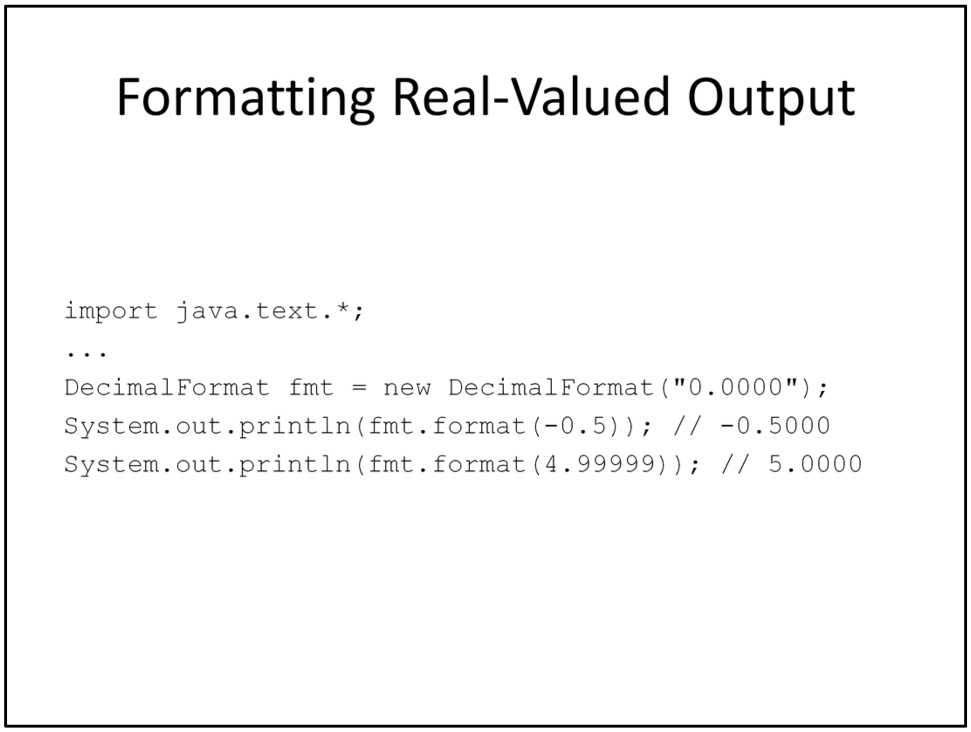
So what's the runtime of this algorithm? The first thing to point out is that the runtime of the bisection algorithm depends on more than just n. It takes order n work to cut the interval in half, since we have to evaluate a polynomial, but the number of cuts we make depends on the size of the interval and the final tolerance we want, NOT the degree of the polynomial. We're gonna call the number of cuts we make P. Notice that this is related to how many digits of precision we want in our answer, which is why we're choosing the letter P here. We can then write out a recurrence relation for our main algorithm. Notice that once we've recursively found the roots of our derivative, we need to run a bisection on each of the up to n intervals we get from our critical values, each of which takes order nP time. If we unroll our recurrence, we get an upper bound that is cubic in n.



ArrayList

```
ArrayList<Double> arr = new ArrayList<Double>();
arr.add(3.0);    // [3.0]
arr.add(4.0);    // [3.0, 4.0]
arr.set(0, 1.0); // [1.0, 4.0]
for (double val : arr) { // loops in order
  System.out.println(val);
}
arr.add(0, -1.0); // [-1.0, 1.0, 4.0]
arr.get(1);       // 1.0
```

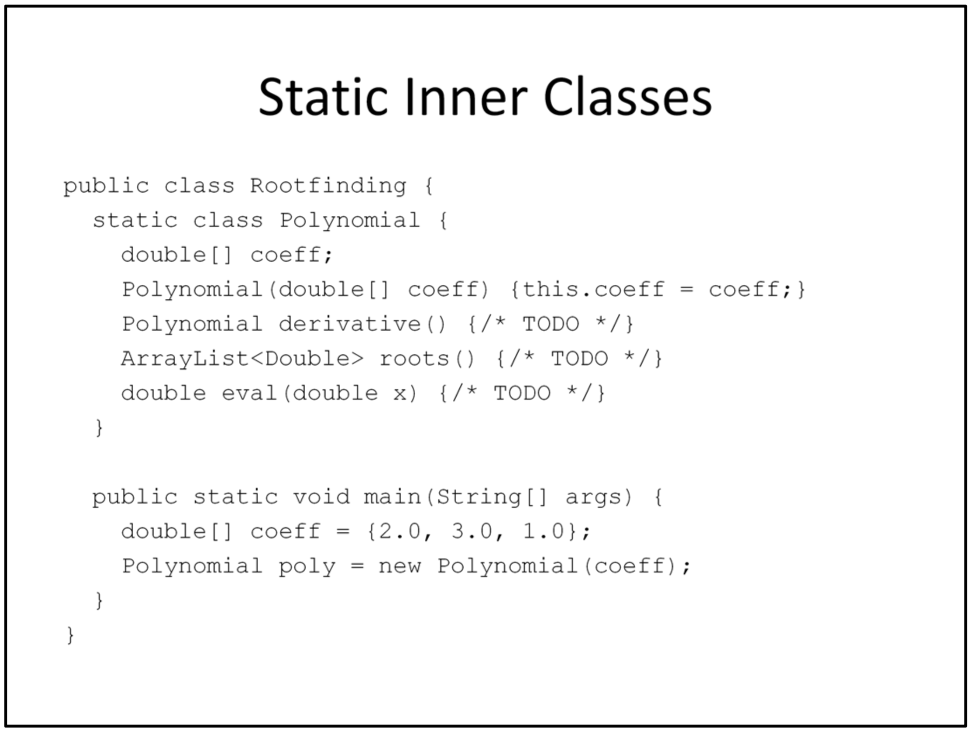
Now for some Java-related stuff. In the first week I brought up the ArrayList and told you folks not to use it for the problems that week. That doesn't mean you should never use ArrayList; in fact, it's gonna come in handy today. The reason for that is it can be really convenient to take advantage of its add method, which inserts an element into the array at the specified location (or the end if no location is given), shifting everything to the right as needed. It's also convenient to iterate over, since the foreach construction works with it. The only thing that's kind of clunky is accessing a specific element, since there's no operator overloading. I've added ArrayList to the Java quick reference document on the course web site if you want to keep that handy while coding.



Formatting Real-Valued Output

```
import java.text.*;
...
DecimalFormat fmt = new DecimalFormat("0.0000");
System.out.println(fmt.format(-0.5)); // -0.5000
System.out.println(fmt.format(4.99999)); // 5.0000
```

Next up is handling I/O for doubles. Reading a double is easy; you just use Scanner.nextDouble() the same way you use Scanner.nextInt(). Writing a double takes a bit more work, especially if you want to format it in a useful way. If you look over the problem statement, you'll see that you always need to print out 4 digits after the decimal place. To do that, you'll use this class called DecimalFormat, which is found in the java.text package. Notice that that means you need to include one more import statement at the top of your program for this week. While there are many different format strings that you can pass to DecimalFormat, the only one you'll need for this class is the one that prints out a fixed number of digits after the decimal place. To do that, follow the example here. Notice that you just need to create one DecimalFormat object, and from then on, you can reuse it every time you want to write out a double in that format.



Static Inner Classes

```
public class Rootfinding {
  static class Polynomial {
    double[] coeff;
    Polynomial(double[] coeff) {this.coeff = coeff;}
    Polynomial derivative() {/* TODO */}
    ArrayList<Double> roots() {/* TODO */}
    double eval(double x) {/* TODO */}
  }

  public static void main(String[] args) {
    double[] coeff = {2.0, 3.0, 1.0};
    Polynomial poly = new Polynomial(coeff);
  }
}
```

One last note: You may find it handy to define a class that represents a Polynomial. The thing is, every problem you submit in this course needs to fit into one file. So if you want to define an additional class, I'd recommend doing so as a static inner class, as you see here. If you do that, you can treat it just like you would any other class in your main code. Notice that in this example we store the coefficients in arrays but the roots in ArrayLists. Can anyone tell me why this is a good idea? (We know the number of coefficients when we create the polynomial, but we don't know the number of roots until after we find them.)