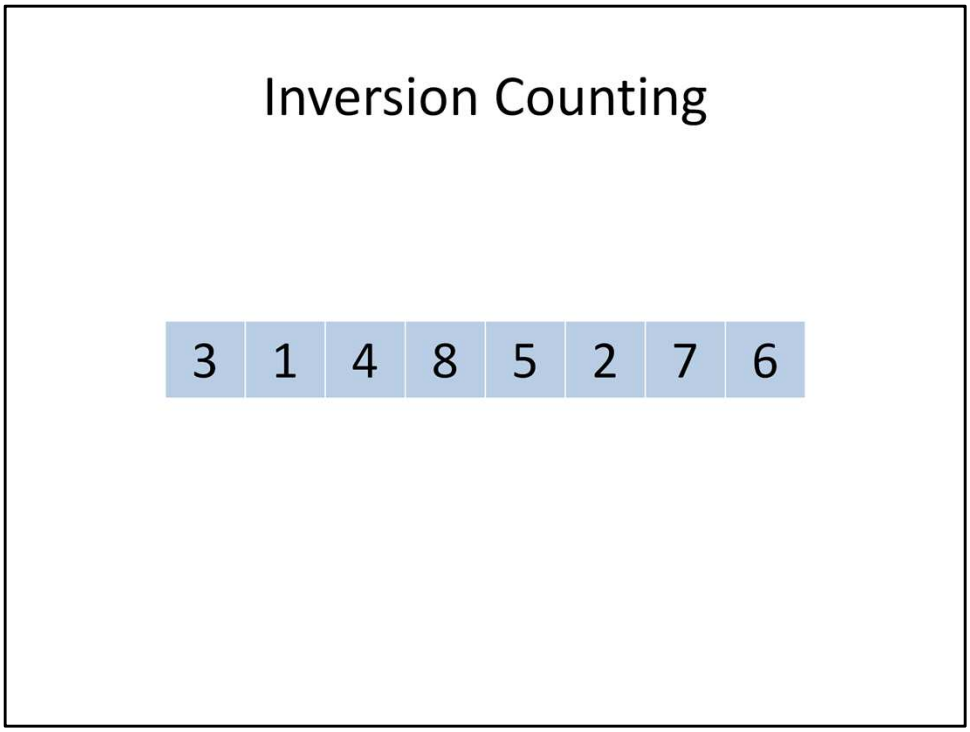
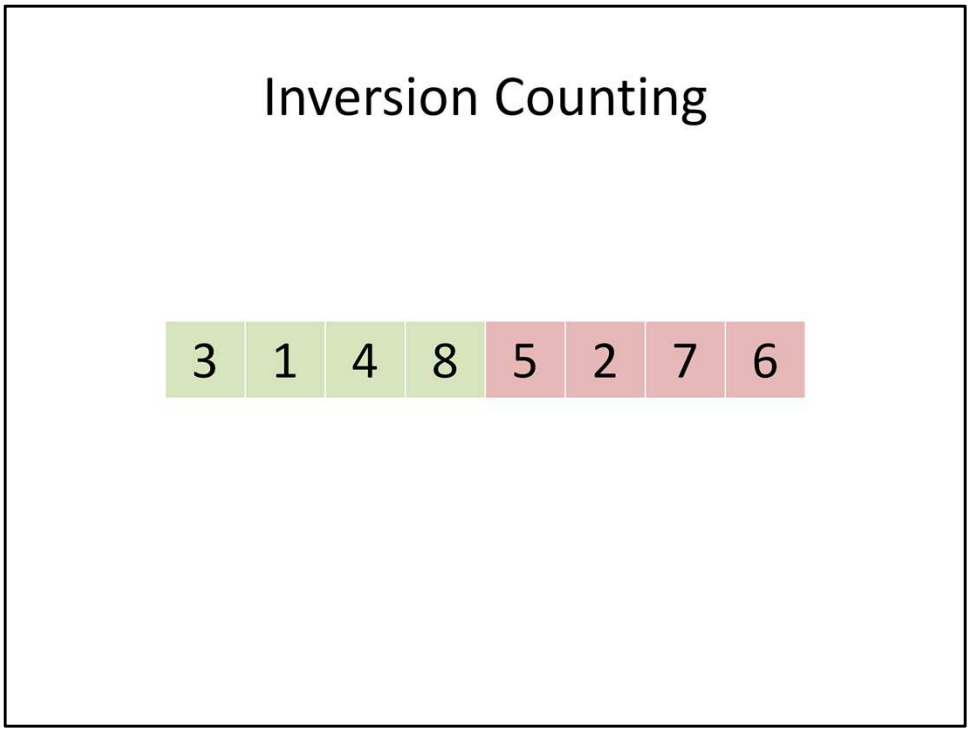
CS161 Programming Section

July 11, 2013

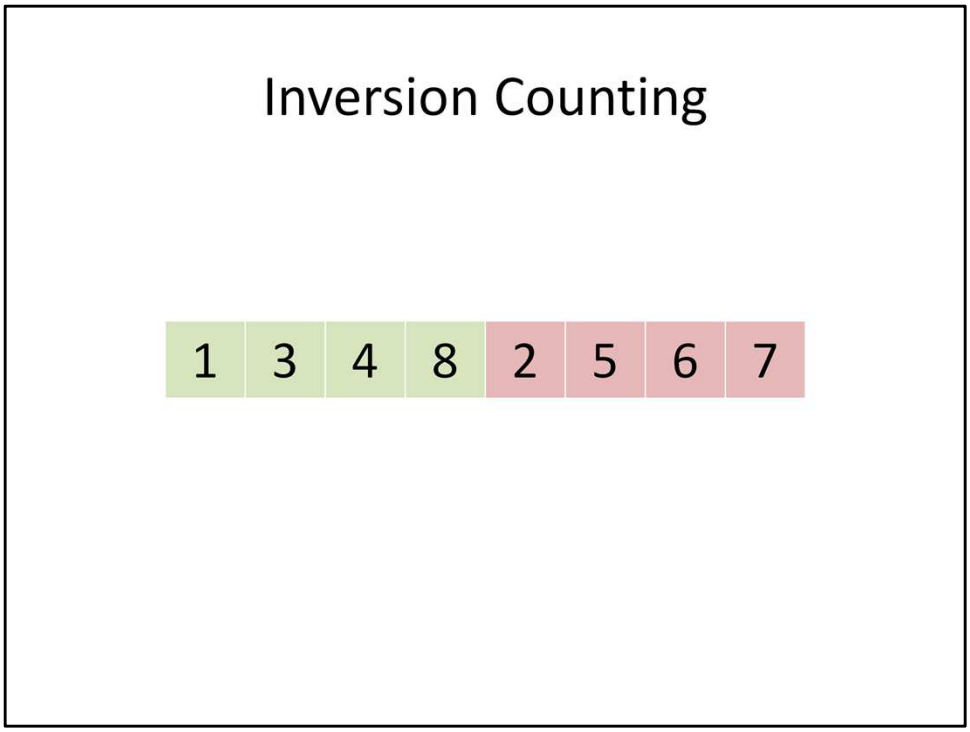
Hi everyone. Starting this week I'm going to make a couple tweaks to how section is run. The first thing is that I'm going to go over all the slides for both problems first, and let you guys code them both up afterwards. That way you can each work at your own pace. I still would recommend solving the first problem before the second one, as the second one is substantially more involved, as it was in the last two sections. The second thing is that the data files are now available on the course website in addition to that AFS folder that I've been pointing you guys to. That should make it easier for you if you want to code on your own machine.



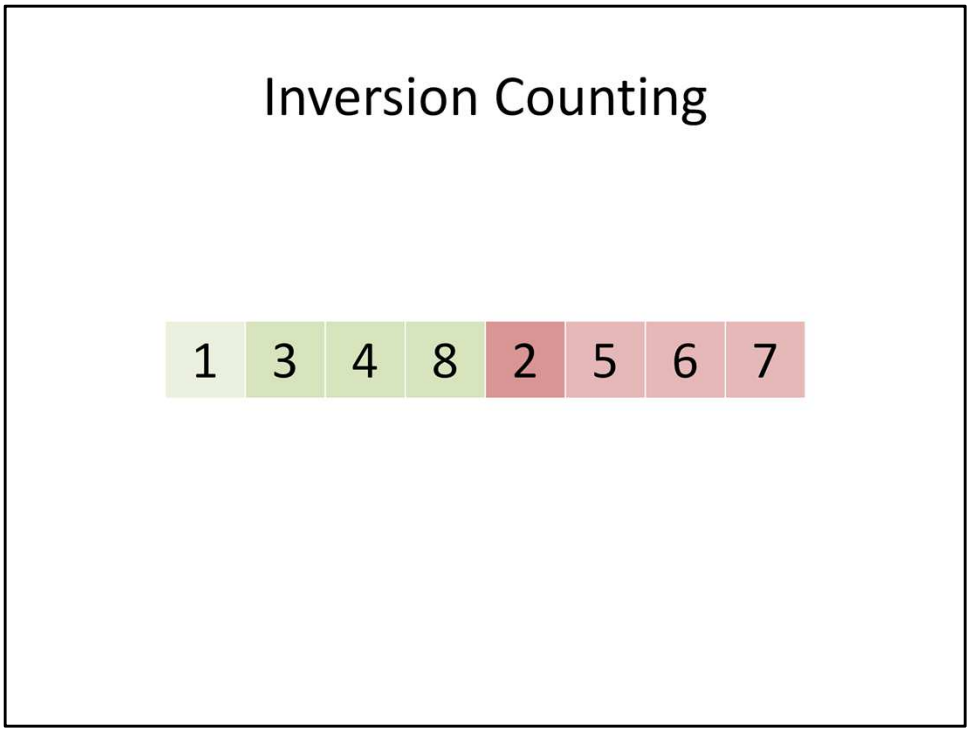
This week we're going to get some practice implementing divide and conquer algorithms. The first algorithm we'll cover is counting the number of inversions in an array of numbers. You might remember from the first day of class that an inversion is a pair of numbers that are out of order. For example, the 3 and the 1 here are an inversion, as are the 8 and the 7. Now if we wanted to count the number of inversions, the first idea that comes to mind is simply checking all pairs of numbers. That will give us an n^2 algorithm. But if we're featuring this problem in the divide and conquer section, then there must be some divide and conquer trick to solving this, right?



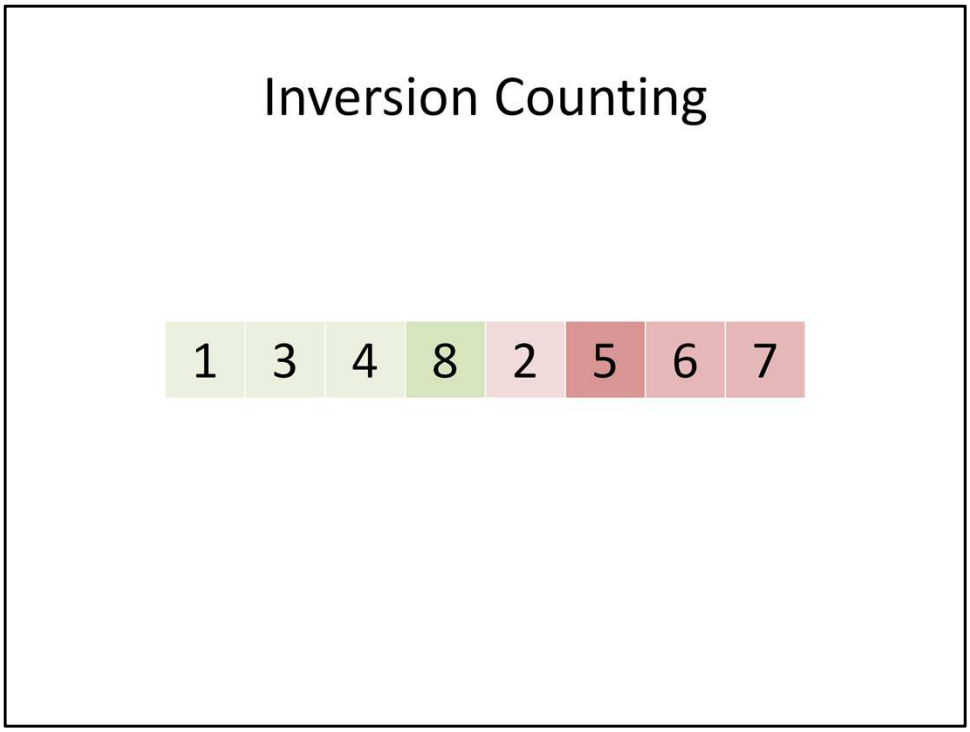
Well, let's try splitting the array down the middle. Let's suppose we recursively compute the number of inversions in the left and right halves of the array. Then the only inversions that are left are the ones where the first element is in the left array and the second element is in the right array, which we'll call green-red inversions to match the picture. But how can we count these? It still looks like we have to check all the pairs. Notice, though, that if we change the ordering of elements within one of the halves, we don't actually change the number of green-red inversions.



So let's sort the two halves. We'll worry about how much this costs later. Does this give us anything useful? Well, let's count the number of green-red inversions that each red element belongs to.



The 2 belongs to 3 green-red inversions, namely with the 3, 4, and 8.



The 5 belongs to 1 inversion, with just the 8. Same goes for the 6 and 7. Notice that for each red element, the green elements that it participates in inversions with, which are the green elements it's smaller than, belong to some range in the green array that starts in the middle and goes all the way to the end. Furthermore, as we advance through the red elements, the range of green elements it's smaller than shrinks. So we could imagine stepping through each of the halves, advancing one pointer or the other depending on which value was smaller, and adding up the size of the remainder of the first half as we go along. But this stepping process should remind you of the merge step of mergesort, because it is EXACTLY that. Remember that we mentioned earlier that we wanted to sort the two halves. That means when we're done with ourselves, our entire array also needs to be sorted. So we're going to have a side effect of our algorithm, which is to sort the array, even though the goal is just to count the number of inversions. So our code is going to look just like mergesort, except we're going to have a running total of the inversions we've seen so far, which we update while we do the merge.

Inversion Counting

- Input
  - A permutation of the numbers from 1 to n
  - n between 1 and 100000 (WARNING: 32-bit overflow)
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+1: 1 integer
- Sample

```
5
1
3
5
2
4
```

- Output
  - The number of inversions of the permutation
- Output Format
  - Line 1: 1 integer
- Sample

```
3
```

Here's the problem specification slide. One thing I'd like to point out is that this time around, the bounds we have WILL cause our answer to overflow a 32-bit integer in the worst case. A useful skill to pick up when coding is to be able to ballpark what the largest values, time, and space usage will be, given the limits on the problem size. Here, the asymptotic bounds we've been computing on the theory side will come in handy, especially when the gap between what we have and what we need is sufficiently large. For example, we know we'll need order n space for our algorithm, which should make us confident that we won't run out of memory since we have multiple gigs of RAM at our disposal, and n can only go up to a hundred thousand. When the gap is smaller, we need to be more careful. For checking whether we'll fit into a 32-bit integer, we have to work out the maximum number of inversions on any test case we'll see, which is going to be 100000 choose 2, which works out to be just shy of 5 billion. A signed 32-bit integer can store up to about 2 billion, and an unsigned 32-bit integer up to about 4 billion, so we're not going to be able to use either. We'll go up a step and use 64-bit integers, which are called longs in Java and long longs in C++.

Vectors (C++)

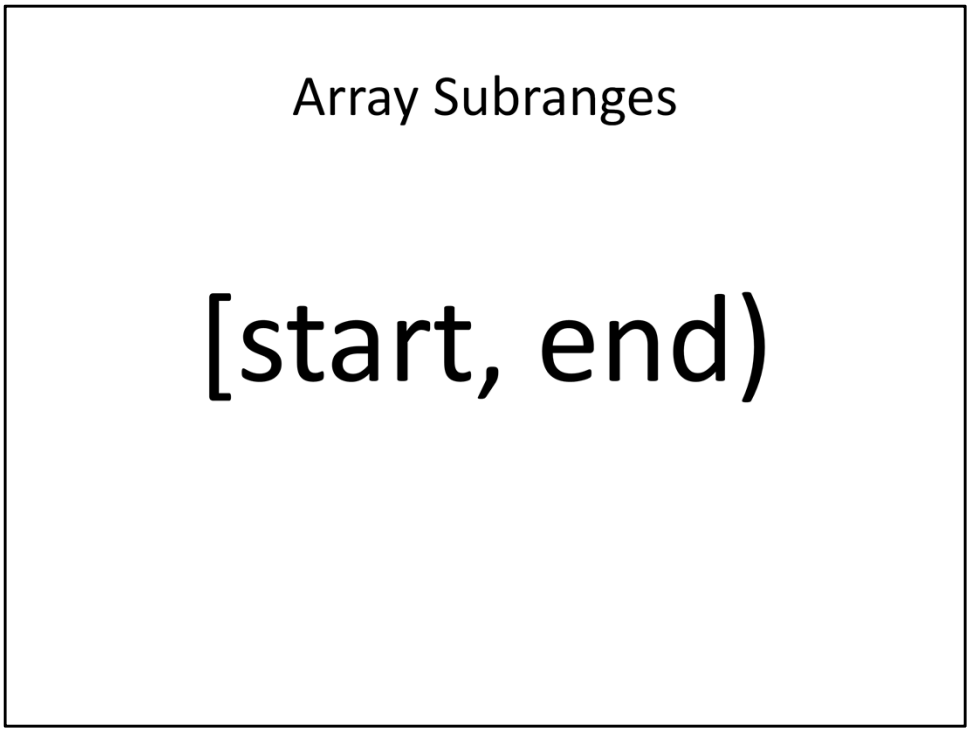
```
#include <vector>

using namespace std;

typedef vector<int> VI;

VI arr(n);
arr[0] = 3;

VI expand;
expand.push_back(3);
```

Now, a detail of mergesort that we've glossed over thus far is that when we perform the merge, we need to make a new array to store the contents of the two arrays we're merging. Once we're done with the merge, we'll need to copy that merged array back into our original array in the appropriate location. This means we'll need to be allocating arrays on the fly. To avoid having to deal with memory management issues in C++, I'd like to remind you guys about the vector class. A C++ vector can be accessed just like an array, but it handles the memory issues for you. It also has this handy little feature where you can tack on another element to the end of the array, and it'll resize itself behind the scenes as necessary. Finally I'd like to point out this typedef line here. This line lets me type VI instead of vector<int> every time I want to work with an array of ints. For one-dimensional arrays, it seems a little frivolous, but once you start having to deal with multidimensional arrays, you'll find that it's much easier to work with a VVVI than a vector<vector<vector<int> > >, especially if you want to preallocate its size.

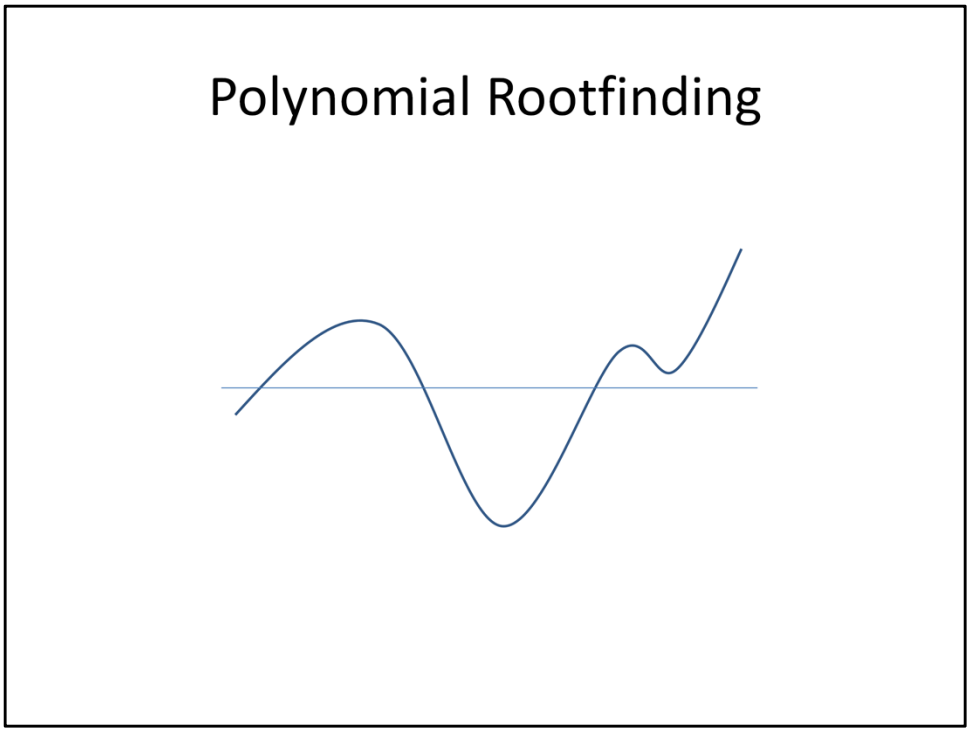
ArrayLists (Java)

```
import java.util.*;

ArrayList<Integer> arr = new ArrayList<Integer>();
arr.set(0, 3);

ArrayList<Integer> expand = new ArrayList<Integer>();
expand.add(3);
```

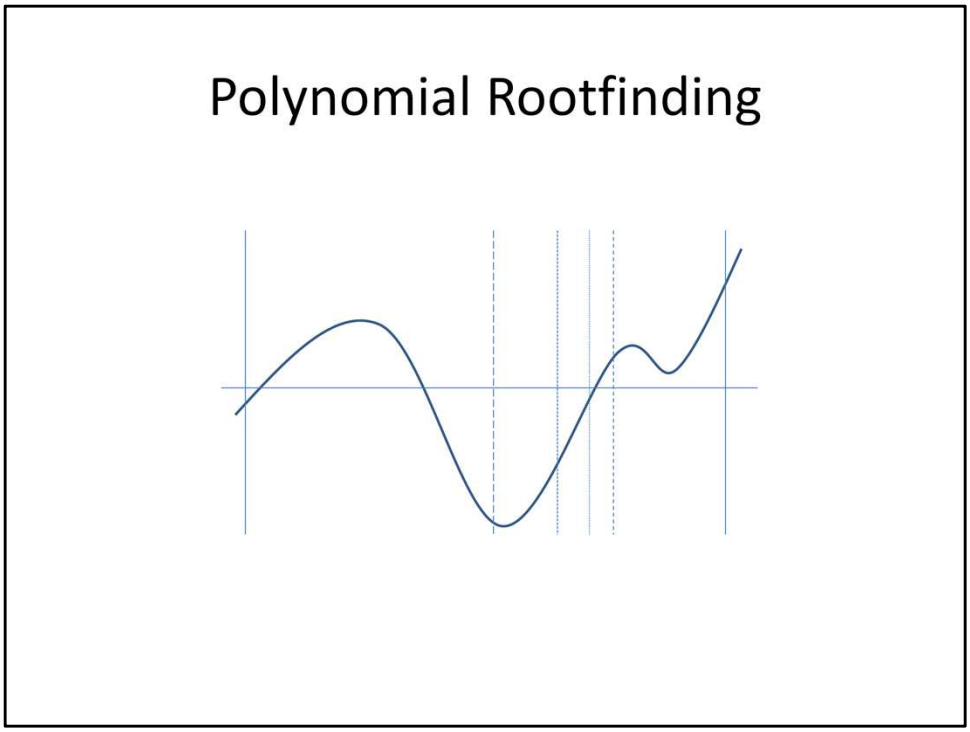
Those of you who are Java coders have an easier time allocating arrays on the fly, so you can continue to use vanilla arrays. If you really want to get access to the expandable array, though, there's a class called ArrayList that works basically like a C++ vector. Unfortunately, there's no operator overloading in Java, so the syntax for working with it is a lot more painful than with a vanilla array. There are also some issues related to what's known as autoboxing in Java when you're trying to make collections of primitive types, which result almost always in performance hits and occasionally even bugs if you forget what autoboxing hides, so in general, unless the expandable array part is really important, you're better off using vanilla arrays.



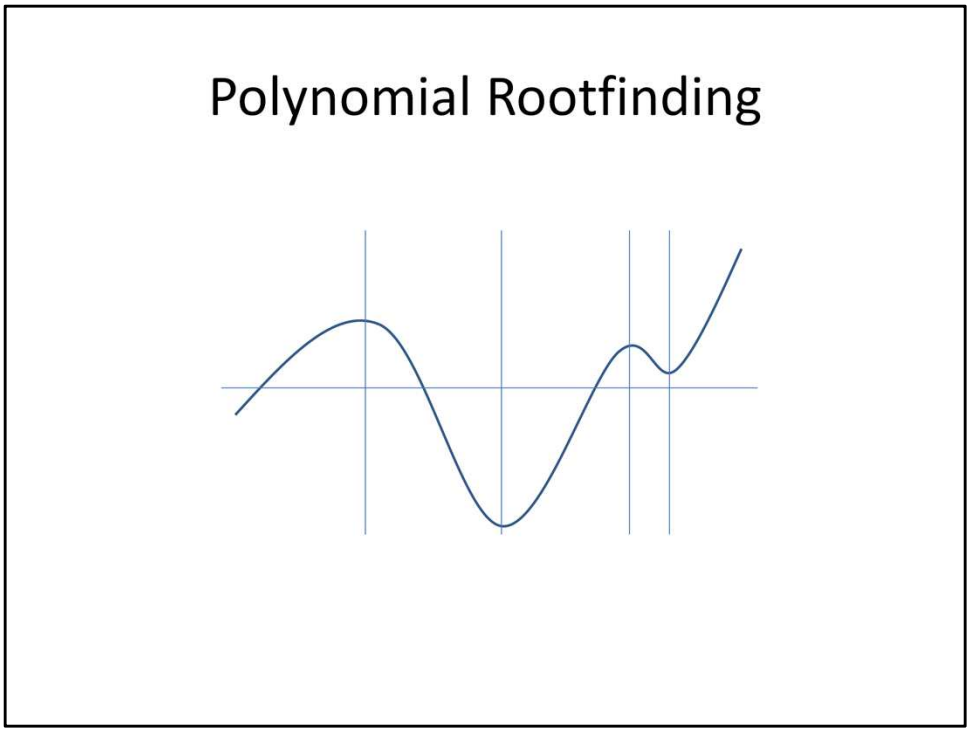
One last hint on coding this problem up. You will probably want to access subranges of your original array in your recursive calls to avoid unnecessarily copying parts of your array. To do this, you'll need to pass in information about the range of the array to access to your recursive function, namely, the start and end. When doing this, there's a convention that's really useful to follow, and that's making start inclusive and end exclusive. This means, for example, that the whole array has a start of 0 and an end of n. Following this convention throughout your code will make it easier to avoid off-by-one bugs.



All right, now to discuss the second 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.



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.



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.

Polynomial Rootfinding

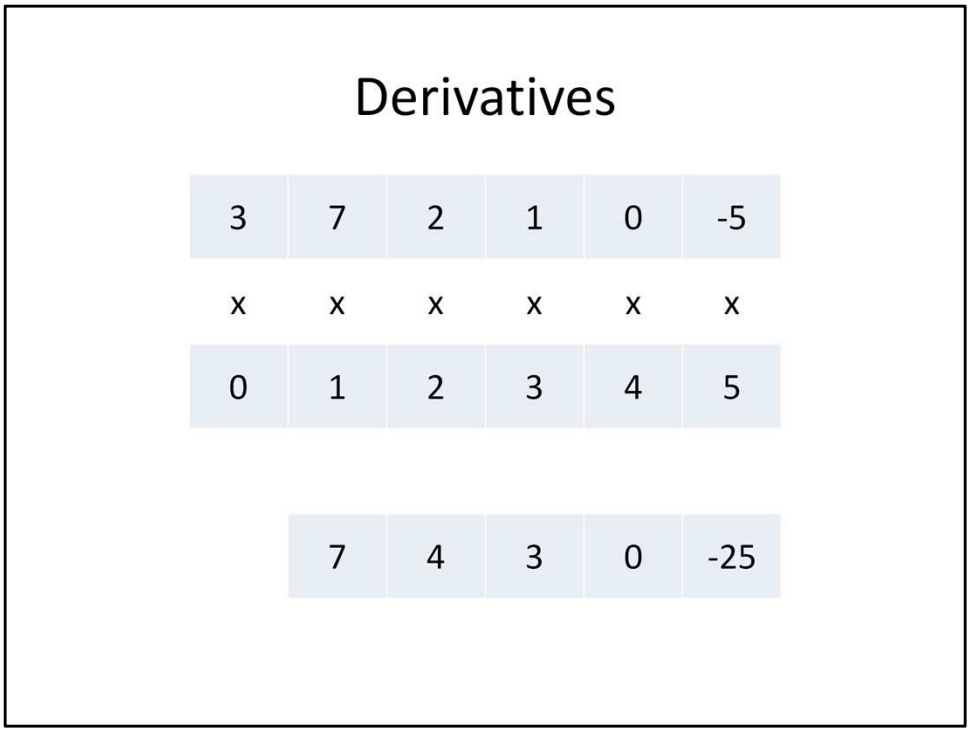
- Input
  - An order n polynomial $a_0+a_1x+a_2x^2+...+a_nx^n$
  - n between 1 and 10
  - no repeated roots (including that of any derivative)
  - all relevant x-values between -1000 and 1000
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+2: 1 double a_i
- Sample

```
2
2.0000
3.0000
1.0000
```

- Output
  - All k real roots of the polynomial, sorted in ascending order
- Output Format
  - Line 1: 1 integer k
  - Lines 2 to k+1: 1 double r_i
- Sample

```
2
-2.0000
-1.0000
```

All right, here's the spec slide for this problem. As promised, the input will be nice enough that you don't need to worry about any edge cases. There will be no repeated roots, and all of the roots will fall within a fairly narrow range so you know where you can start and end your search. Notice that we list the coefficients of the polynomial starting with the constant term and ending with the highest degree term. Also remember that a degree n polynomial has n + 1 coefficients. For this problem, I'll recommend representing a polynomial as an array of coefficients.



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

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$$

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

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 = log(intervalwidth / gap)
  - related to # of bits 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 bits 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.