CS161L: Implementation of Algorithms

Friday, April 4, 2014

Hello everyone, and welcome to CS161L, where you'll get a chance to implement a lot of the algorithms covered in CS161. My name is Andy, and I'll be your instructor for this course. I'm also a CA for the regular CS161 class, and I'll be responsible for managing the final project in that class, so this course will be good preparation for that project. A quick announcement: I will be out of town next week, but I've arranged for substitutes to take my place in both sessions next week, so we will still be having class as planned.

Prerequisites

- Familiarity with C++ or Java (CS106 level)
- Basic knowledge of Linux command line (managing files and directories)
- Co-requisite: CS161

Now, this course is designed to be a companion class to 161, which means that 106 is an indirect prerequisite. What this means for this section is that we're going to assume that you're comfortable writing basic programs in C++ or Java, and have at least some idea of how to use their standard libraries. Also, since section is being held in this computer lab, where Linux is what is installed on all the machines, we're going to assume that you have or can pick up some basic knowledge of the Linux command line. If you made it through the Getting Started documents on the course web site without too much trouble, you should be ok for this class. Speaking of, who here has actually worked through the Getting Started documents and gotten their machine set up?

Format

- Before Friday: look over problems
  - Course web site: http://cs161l.stanford.edu
- Friday, first 15 minutes: go over problem solutions/coding techniques
- Friday, remaining hour: code up everything (ideally: no homework!)
- By end of Monday: submit code

Now, how will this course be conducted? Well, every Wednesday, I'll post the problems for the week. I'd like you all to look over the problems before the session on Friday so you at least have a sense of what we'll be covering. During the session, we'll spend the first 15 minutes or so discussing the problems, talking about coding techniques or pitfalls relevant to the problems, and pointing out what you should be observing while writing and running your code. After that, the remainder of the time will be devoted to writing code. If you come across any issue while coding, speak up, and we'll work through it together. I've designed the problems with the intent that most of you finish coding them in this hour, so once you're done, you can submit, and you don't have to worry about homework. Sometimes debugging can take a bit longer than planned, though, so the deadline to submit isn't until the end of the following Monday.

Submissions

- Groups of size 1 to 3 (2 recommended)
- Java officially supported (C++ unofficially supported)
- Each week: 1 required problem (7/9 to pass)
- Honor Code: Your code must be your own!

Now, just like in CS161, you're allowed to work in groups of up to 3 people. Personally, I'd recommend pairing off, as in my experience having one person at the keyboard and the other person double checking what's typed seems to be most effective. This class only officially supports Java, which means that any code that shows up on these slides will be written in Java. Now, I know that CS106B and CS106X are both conducted in C++, so many of you are more comfortable in C++ than in Java. For the most part, the problems we cover in this class will use only basic language features, which means the C++ code and the Java code won't look all that different, so if you can write the former, you can write the latter. That said, if you really want to code in C++, I will accept submissions in that language, as long as they meet all of the problem requirements. Each week, we may cover multiple problems, but only one of those problems will be required. I still recommend working through all of them, because often the non-required ones will be useful for understanding the required one. In order to pass this course, you need to solve at least 7 weeks' worth of problems. This means you can skip 2 problems if you want to. Notice that there's no late policy in this class though; I will not accept late submissions under any circumstances. Also, a lot of the algorithms that we're going to cover in this course are classic algorithms. That means that if you were to Google search for any of these problems, you could almost certainly find code that does exactly what you want. But that defeats the purpose of this course, which is to give YOU practice in writing common algorithms. So please don't do that; write all the code yourself.

Course under Construction

- Feedback greatly appreciated!
  - My email: tanonev@stanford.edu
  - Find me after class or during my CS161 OH Mondays from 3 to 6
  - Mid-quarter anonymous survey

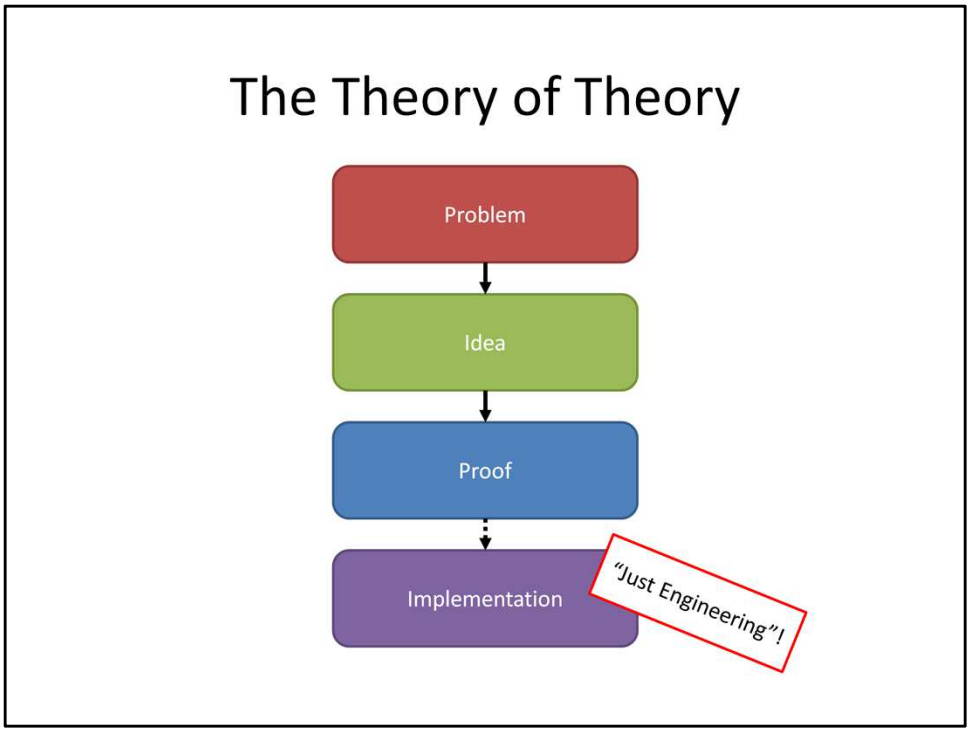
One last note before we begin with the course material: This is a new course, so I'd love to hear any thoughts you have on how the course should be run. If you have thoughts on the pacing of the course, or if there's an algorithm you really want to see covered, or whatever, come talk to me after class, or shoot me an email. If you're not comfortable talking with me directly, I will be sending out a survey halfway through the course that's completely anonymous.

Why Code Algorithms?

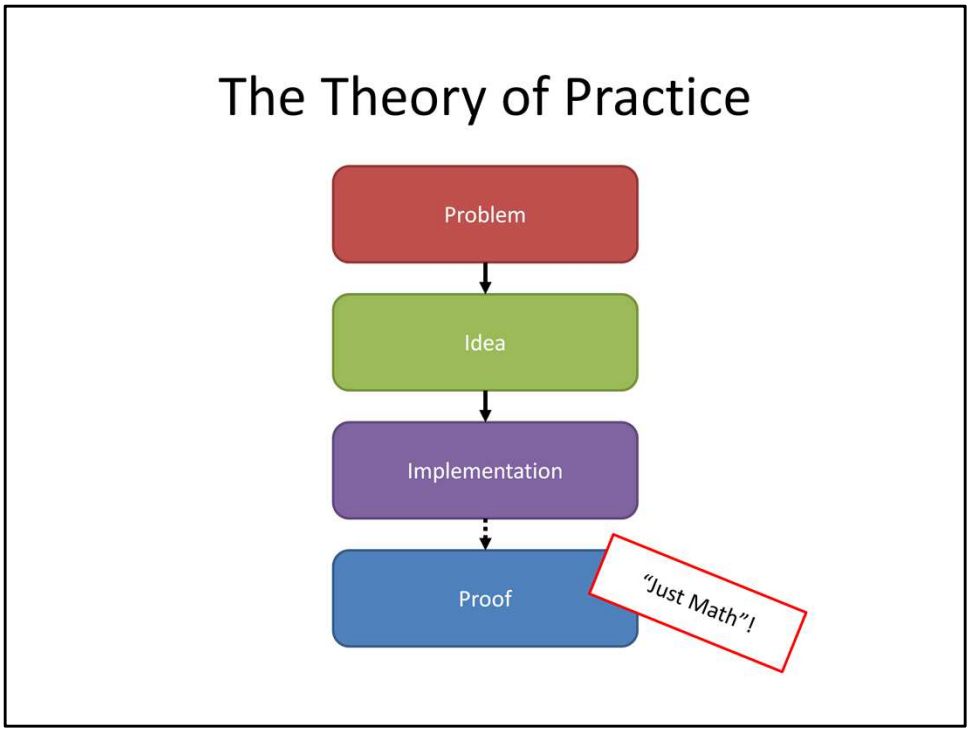
"Beware of bugs in the above code; I have only proved it correct, not tried it."

--Don Knuth

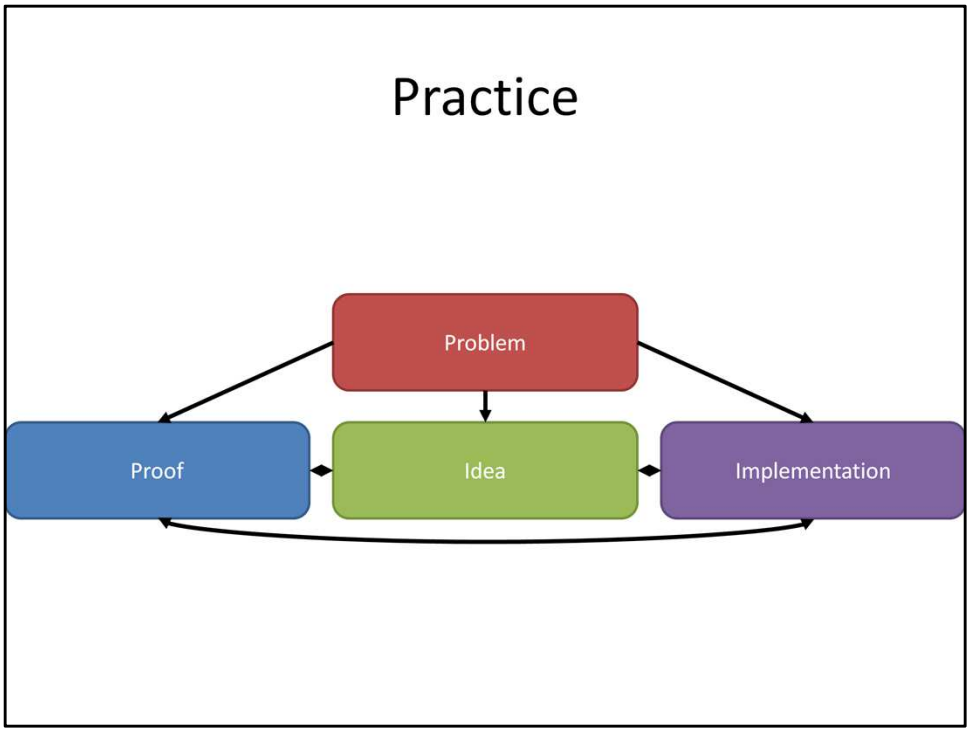
All right, let's get started. As you know, CS161 is a theory course, so the first thing we should do is ask ourselves, "Why, in a theory class, do we care about coding up the algorithms we learn? Isn't that part just mechanical?" Well, that's a fair question. After all, as this quote by the famed Donald Knuth would suggest, it IS often the case that we'll stop after we've successfully discovered our algorithm.



See, there's this ideal approach to problem solving that computer science theory courses would have you believe. You're given a problem, which you think about for a while until you come up with an idea to solve it. You then formalize your idea and prove its correctness, and then at that point you have an algorithm that you know would work if you ever sat down and coded it up. Everything after that is "just engineering".



But if you were to ask around a bit more, you would find plenty of people who would tell you that when you have an idea, you should code it up and see how it works. After all, if it works well, you can use it as is and worry about why it works later; and if it doesn't work, then your proof wouldn't have gone anywhere anyway. Once you have code that backs up your idea, proving it works is "just math". Now, this shouldn't surprise us that there are these two trains of thought; after all, we're computer scientists, and we can think of computer science as living halfway between math and engineering. The thing is, there's something wrong with both of these pictures. Note that in both cases, we have this magic step where we come up with an idea for how to solve the problem. How does this step even work? Is it inspiration? Do you just bang your head against the wall until a solution comes to mind or you run out of brain cells?

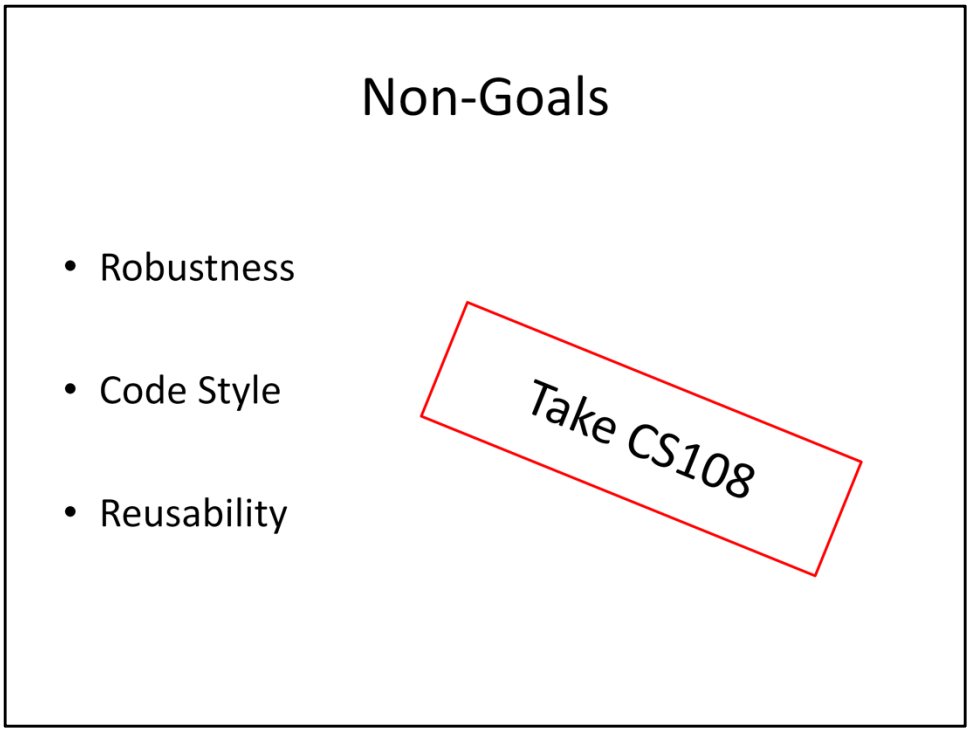


The thing is, these ideas don't come from nowhere; in fact, when we're trying to solve a problem, often we'll have to attack at it from all sides, hoping that some direction will allow us to make some progress. Sometimes we might write some experimental code to see if we can discover some patterns in the problem that we can exploit. Other times we'll research similar problems and read through their proofs to see if anything there applies to our problem. All of these parts feed back into each other, so even if all we care about is finding a provably correct solution, we can still benefit from coding up our ideas.

Goals

- Turn algorithm descriptions into real code
- Efficiently write code
- Write efficient code
- Learn theory from code

So here's what I'm hoping you'll get out of this course. First, we're going to spend a good chunk of time turning the descriptions of algorithms that we come across in class into actual runnable code. Often we'll discover that there are some nasty implementation details that we shoved under the rug in the process of our high-level description. We'll also take a look at some algorithms not covered in class. Next, we'll cover some tricks for being able to write what we call "scratch code" quickly and easily. In this course, we're going to focus on using code as a tool for understanding the algorithm, so the emphasis will be on how to do so with minimal effort. We're also going to spend some time covering practices that will make our code more efficient, even if it's just by a constant factor that would get swallowed up in big-O analysis. And finally, we're going to use some coding experiments to motivate refining our theoretical understanding of a problem to better fit what we observe in practice.



Before we get going, let's just briefly mention what we WON'T be covering in this course. We are NOT going to be writing industrial strength code. That means, for example, that we will assume that our input is always well-formed, and we'll handle input in a way that's convenient for us, even if it's not scalable. We're also not going to place any focus on coding style, nor are we going to spend time designing our code in such a way that we'll be able to plug it into other projects easily. Don't get me wrong, these are all valuable skills, and if you want to pick these skills up, there's a great course called CS108 that will teach them to you. But that's not this course.

Warmup: Insertion Sort

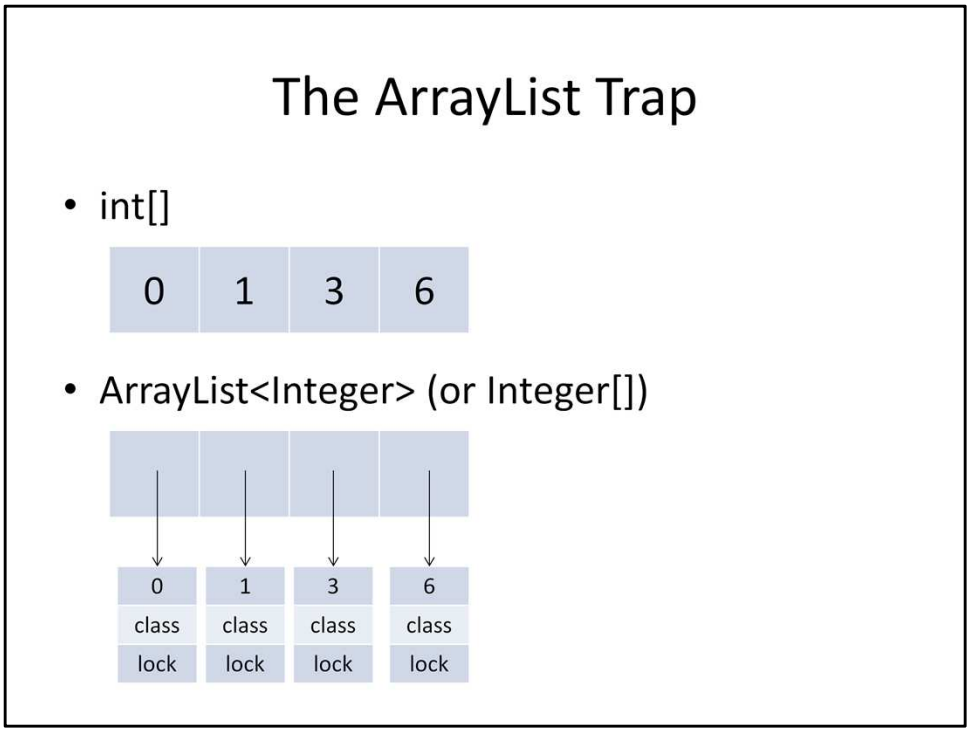
- Refer to 161 slides or CLRS for details
- Use the tutorial program as a starting point
  - Modify program to read multiple test cases
  - Replace the library sort call with your own implementation

All right, now to go over the problems we'll be solving today. The first two problems are warmups of sorts, to get you familiar with the format we'll we be using throughout the quarter. The problem statements will always be posted on the web site in Google doc form, and you'll wanna have them handy for when you're coding the problems up. I'm not going to repeat material that's covered in 161 proper, so if you need to look up how insertion or mergesort work, you should refer to the 161 slides or textbook. The code that the Getting Started document had you write is a great starting point for both problems. For insertion sort in particular, all you need to do is modify the main method to read multiple test cases instead of a single one, and modify the sort method to use insertion sort instead of calling the library sort.

Warmup: Mergesort

- Refer to 161 slides or CLRS for details
- Convention is king!
  - mutate input (sub)array
  - array ranges: [start, end)

Mergesort is a little trickier to code, though I've been told that implementing mergesort is a standard 106 assignment, so this should be review for most of you. Again, refer to the 161 materials if you need them. One thing to keep in mind when working with arrays in a more involved algorithm is to pick a convention and stick to it. Write the convention down somewhere, and every time you write a bit of code, make sure it conforms to that convention. This is especially useful when you're trying to figure out what the pre and post conditions of a recursive function are going to be. For this problem, I'm going to recommend that your recursive sort function take in the whole array, as well as a range, and that the function mutates the input array so that that range is guaranteed to be sorted at the end of the function. Note that this does NOT mean that you have to do the merging in place; in fact, you should NOT try to do that. Instead, you should perform the merge using a new array you allocate, and then copy the merged data back into the original array at the end. Another convention that's useful to set is whether range bounds are inclusive or exclusive. The convention I prefer is for the start to be inclusive and the end to be exclusive. If you try it out, you'll see that this convention keeps you from having to add or subtract 1's from bounds for a lot of things you want to do, like split an array in half.

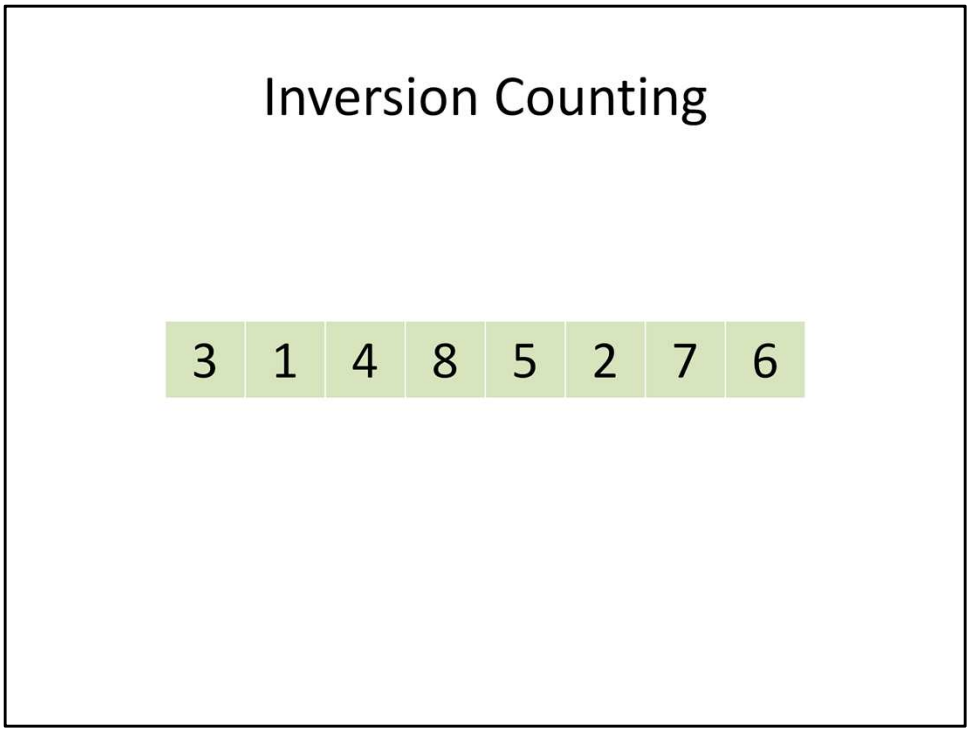


One last hint on coding this problem up. When you're doing the merge part of mergesort, you might be tempted to look for some class that handles the "add this to the end of the list" step nicely. If you look around, you'll find this very standard class called ArrayList that does just that. An ArrayList is a Java Collection that works like a resizable array, allowing you to append to the end of an array and automatically managing the size behind the scenes, much like a vector in C++. This can be very convenient, but it comes at a performance cost, especially when you try to use it in place of a primitive array. Why is that? You can't actually use Java Collections with primitives directly; instead, there are these wrapper objects for all of the primitive types. There's this thing called autoboxing which allows you to use int and Integer basically interchangeably, but that hides the fact that working with Integers is much more expensive than working with ints. So in case you ever wondered why some Java program uses 10 times as much memory as a C++ program that looks almost identical, this is a pretty common reason why.

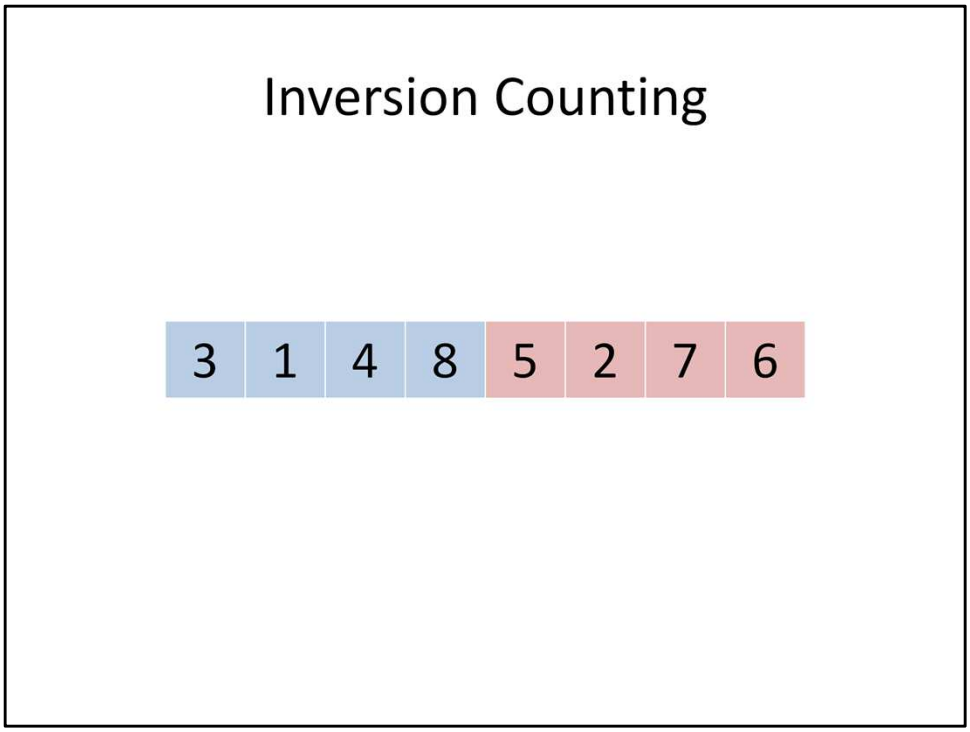
Why Both?

- Practical performance tradeoffs
- Improve confidence in answer
  - Insertion sort easier to understand and to code
  - Check your mergesort output against your insertion sort output on `medium.in`
  - If they don't match, error more likely in mergesort
  - If they do, you can run mergesort on `large.in`

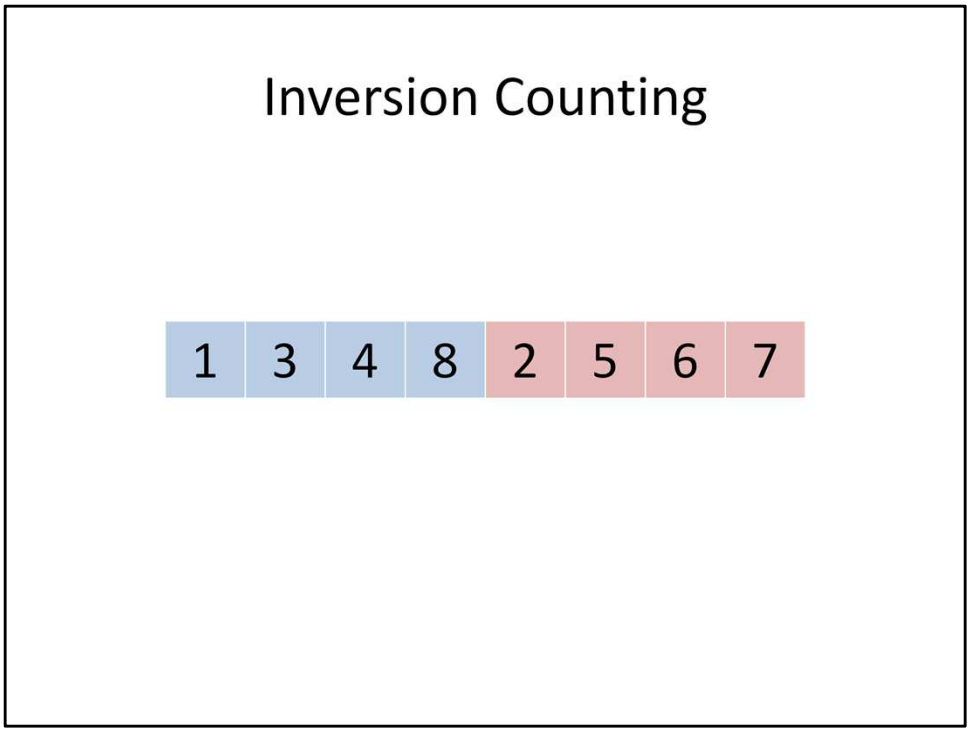
Now, why are we coding up both insertion sort and mergesort when we know one is asymptotically better than the other? Well, as mentioned in the 161 lecture, asymptotic performance doesn't necessarily correspond to practical performance, especially when the size of the input is limited. But the main thing I want to focus on is a general technique for verifying whether you've successfully coded up a tricky algorithm. Insertion sort is a lot easier to understand and to code up than mergesort. If you look at the data files I've provided you, I haven't actually given you the answers to the larger input. You're going to have to figure out those answers for yourself. The way you do this is you run insertion sort on the medium file to generate ground truth. Then you run mergesort on it to see whether you get the same answer. Once you're satisfied that they match, then you can move to the large file, which is too big for insertion sort to process in a reasonable amount of time.



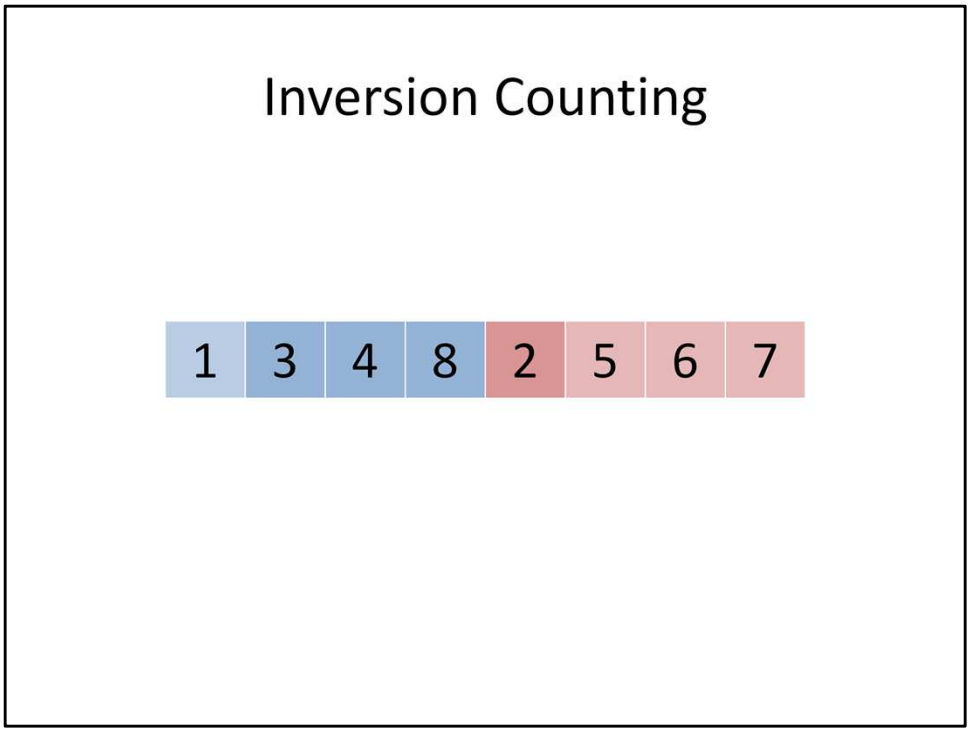
Now let's move on to the main problem for this week, which connects insertion sort to merge sort. You might remember from the first day of class that insertion sort has a wide gap between its best case run time and its worst case run time, depending on the input. Let's see if we can find some way of explaining this as a property of the input, independent of the algorithm. We know that the runtime of insertion sort is closely tied to the number of swaps it takes to sort the array. But talking about swaps is still connecting the input to the specific algorithm. Instead, we're going to look at this thing called an inversion. An inversion in an array is a pair of elements that are out of order. For example, the 3 and the 1 here are an inversion, as are the 8 and the 7. Notice that this definition doesn't depend on any algorithm; it's intrinsic to the array itself. What kind of an array has the smallest possible number of inversions? (Sorted.) How many does it have? (0.) What kind of an array has the largest possible number of inversions? (Reverse sorted.) How many does it have? (n choose 2). Now, what does this have to do with insertion sort? Well, each time we do a swap in insertion sort, we reduce the number of inversions in the array by exactly one, since we only swap adjacent numbers. Also, we stop when the array is sorted, which is when we're out of inversions. That means that the number of swaps we do is exactly the same as the number of inversions in the original array. Now suppose I asked you to count the number of inversions in an array. One thing you could do is run insertion sort and count how many swaps it makes. But can we do better?



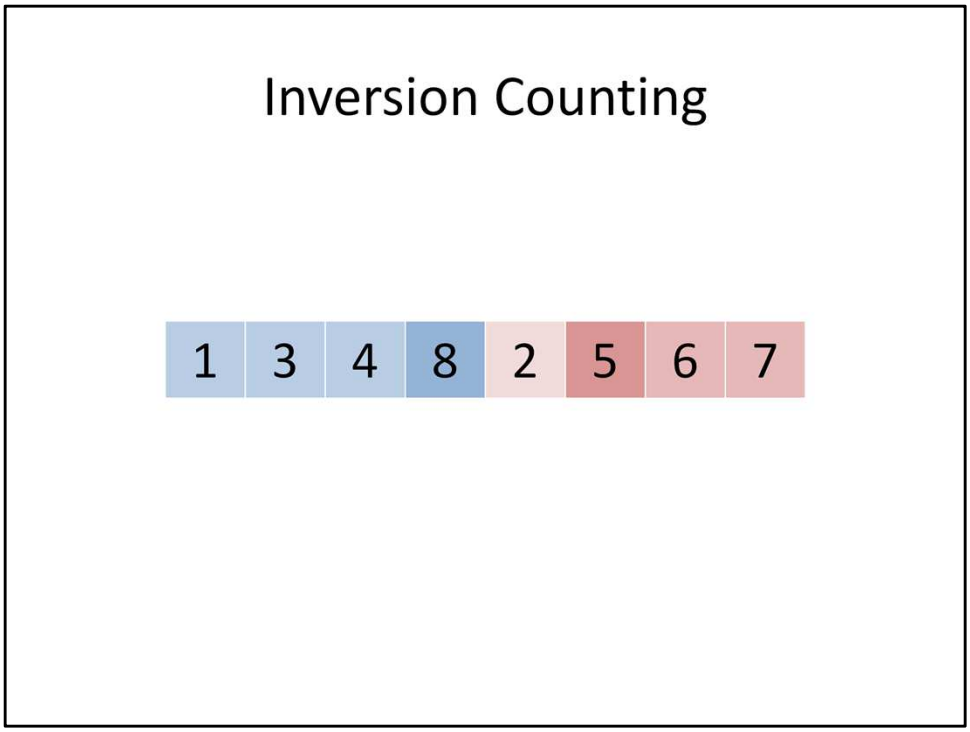
Of course we can; I wouldn't ask otherwise. Let's try a divide-and-conquer approach to this problem. So we'll start by splitting the array down the middle. Let's suppose we recursively compute the number of inversions in the left and right halves of the array. Then the only inversions that are left are the ones where the first element is in the left array and the second element is in the right array, which we'll call blue-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 blue-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 blue-red inversions that each red element belongs to.



The 2 belongs to 3 blue-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 blue elements that it participates in inversions with, which are the blue elements it's smaller than, belong to some range in the blue 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 blue 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 what we're going to do is we're going to modify mergesort to track the number of inversions in the array, sort of like how we first considered modifying insertion sort to track the inversions in the array.

Check Your Work!

- Use insertion sort for ground truth
- Check *output* bounds, not just input bounds

Now, just like when you were working on sorting, you can use your insertion sort implementation to have a trustworthy count of the number of inversions for the medium input file, and then use that to check the correctness of your modified mergesort. There is one more detail that you need to worry about, though. The problem statements will have input bounds which you can use to determine what is or is not a reasonable approach. For example, if the largest number that could appear in the array to sort were bigger than about 2 billion, you'd need to use a long instead of an int. But when coding up a problem like this, it's not enough to just check the input numbers. You also have to check how big any number that is produced can be. The problem specification for inversion counting says there can be up to a hundred thousand numbers. Now remember we said an array of size n could have up to n choose 2 inversions. What's a hundred thousand choose 2? (~5 billion.) So can you use an int to store the answer? (No.) Make sure your function returns a long instead of an int. For this assignment, I've given you the expected output on the large input so you can check your work, including a case that would fail if you used an int instead of a long; that won't necessarily be true in the future, though, so it's important to think through the bounds that are given for the problems. All right, that's all the slides I have for today; the Getting Started on the Myth Cluster document on the course web site has the instructions for how you can access this data, and then you can begin on today's problems. If you have any questions, I'll be right here to answer them.