MIT OpenCourseWare
http://ocw.mit.edu

6.00 Introduction to Computer Science and Programming, Fall 2008
Transcript – Lecture 10

PROFESSOR: Last time we were talking about binary search and I sort of left a promise to you which I need to pick up. I want to remind you, we were talking about search, which is a very fundamental thing that we do in a whole lot of applications. We want to go find things in some data set. And I'll remind you that we sort of a separated out two cases. We said if we had an ordered list, we could use binary search. And we said that was log rhythmic, took log n time where n is the size of the list.

If it was an unordered list, we were basically stuck with linear search. Got to walk through the whole list to see if the thing is there. So that was of order in. And then one of the things that I suggested was that if we could figure out some way to order it, and in particular, if we could order it in n log n time, and we still haven't done that, but if we could do that, then we said the complexity changed a little bit. But it changed in a way that I want to remind you.

And the change was, that in this case, if I'm doing a single search, I've got a choice. I could still do the linear case, which is order n or I could say, look, take the list, let's sort it and then search it. But in that case, we said well to sort it was going to take n log n time, assuming I can do that.

Once I have it sorted I can search it in log n time, but that's still isn't as good as just doing n. And this led to this idea of amortization, which is I need to not only factor in the cost, but how am I going to use it? And typically, I'm not going to just search once in a list, I'm going to search multiple times. So if I have to do k searches, then in the linear case, I got to do order n things k times. It's order k n.

Whereas in the ordered case, I need to get them sorted, which is still n log n, but then the search is only log n. I need to do k of those. And we suggested well this is better than that. This is certainly better than that. m plus k all times log n is in general going to be much better than k times n. It depends on n and k but obviously as n gets big, that one is going to be better.

And that's just a way of reminding you that we want to think carefully, but what are the things we're trying to measure when we talk about complexity here? It's both the size of the thing and how often are we going to use it? And there are some trade offs, but I still haven't said how I'm going to get an n log n sorting algorithm, and that's what I want to do today. One of the two things I want to do today.

To set the stage for this, let's go back just for a second to binary search. At the end of the lecture I said binary search was an example of a divide and conquer

algorithm. Sort of an Attila the Hun kind of approach to doing things if you like. So let me say -- boy, I could have made a really bad political joke there, which I will forego, right.

Let's say what this actually means, divide and conquer. Divide and conquer says basically do the following: split the problem into several sub-problems of the same type. I'll come back in a second to help binary searches matches in that, but that's what we're going to do. For each of those sub-problems we're going to solve them independently, and then we're going to combine those solutions.

And it's called divide and conquer for the obvious reason. I'm going to divide it up into sub-problems with the hope that those sub-problems get easier. It's going to be easier to conquer if you like, and then I'm going to merge them back. Now, in the binary search case, in some sense, this is a little bit trivial. What was the divide? The divide was breaking a big search up into half a search. We actually threw half of the list away and we kept dividing it down, until ultimately we got something of size one to search. That's really easy.

The combination was also sort of trivial in this case because the solution to the sub-problem was, in fact, the solution to the larger problem. But there's the idea of divide and conquer. I'm going to use exactly that same ideas to tackle sort. Again, I've got an unordered list of n elements. I want to sort it into a obviously a sorted list. And that particular algorithm is actually a really nice algorithm called merge sort. And it's actually a fairly old algorithm. It was invented in 1945 by John von Neumann one of the pioneers of computer science. And here's the idea behind merge sort, actually I'm going to back into it in a funny way.

Let's assume that I could somehow get to the stage where I've got two sorted lists. How much work do I have to do to actually merge them together? So let me give you an example. Suppose I want to merge two lists, and they're sorted.

Just to give you an example, here's one list, 3121724 Here's another list, 12430. I haven't said how I'm going to get those sorted lists, but imagine I had two sorted lists like that. How hard is it to merge them? Well it's pretty easy, right? I start at the beginning of each list, and I say is one less than three? Sure. So that says one should be the first element in my merge list.

Now, compare the first element in each of these lists. Two is less than three, so two ought to be the next element of the list. And you get the idea. What am I going to do next? I'm going to compare three against four. Three is the smallest one, and I'm going to compare four games twelve, which is going to give me four. And then what do? I have to do twelve against thirty, twelve is smaller, take that out. Seventeen against thirty, twenty-four against thirty And by this stage I've got nothing left in this element, so I just add the rest of that list in.

Wow I can sort two lists, so I can merge two lists. I said it poorly. What's the point? How many operations did it take me to do this? Seven comparisons, right? I've got eight elements. It took me seven comparisons, because I can take advantage of the fact I know I only ever have to look at the first element of each sub-list. Those are the only things I need to compare, and when I run out of one list, I just add the rest of the list in.

What's the order of complexity of merging? I heard it somewhere very quietly.

STUDENT: n.

PROFESSOR: Sorry, and thank you. Linear, absolutely right? And what's n by the way here? What's it measuring?

STUDENT: [UNINTELLIGIBLE]

PROFESSOR: In both lists, right. So this is linear, order n and n is this sum of the element, or sorry, the number of elements in each list. I said I was going to back my way into this. That gives me a way to merge things. So here's what merge sort would do.

Merge sort takes this idea of divide and conquer, and it does the following: it says let's divide the list in half. There's the divide and conquer. And let's keep dividing each of those lists in half until we get down to something that's really easy to sort. What's the simplest thing to sort? A list of size one, right? So continue until we have singleton lists.

Once I got a list of size one they're sorted, and then combine them. Combine them by doing emerge the sub-lists. And again, you see that flavor. I'm going to just keep dividing it up until I get something really easy, and then I'm going to combine. And this is different than binary search now, the combine is going to have to do some work.

So, I'm giving you a piece of code that does this, and I'm going to come back to it in the second, but it's up there. But what I'd like to do is to try you sort sort of a little simulation of how this would work. And I was going to originally make the TAs come up here and do it, but I don't have enough t a's to do a full merge sort. So I'm hoping, so I also have these really high-tech props. I spent tons and tons of department money on them as you can see.

I hope you can see this because I'm going to try and simulate what a merge sort does. I've got eight things I want to sort here, and those initially start out here at top level. The first step is divide them in half. All right? I'm not sure how to mark it here, remember I need to come back there. I'm not yet done. What do I do? Divide them in half again.

You know, if I had like shells and peas here I could make some more money. What do I do? I divide them in half one more time. Let me cluster them because really what I have, sorry, separate them out. I've gone from one problem size eight down to eight problems of size one.

At this stage I'm at my singleton case. So this is easy. What do I do? I merge. And the merge is, put them in order. What do I do next? Obvious thing, I merge these. And that as we saw was a nice linear operation. It's fun to do it upside down, and then one more merge which is I take the smallest elements of each one until I get to where I want.

Wow aren't you impressed. No, don't please don't clap, not for that one. Now let me do it a second time to show you that -- I'm saying this poorly. Let me say it again. That's the general idea. What should you see out of that? I just kept sub-dividing down until I got really easy problems, and then I combine them back.