MIT OpenCourseWare
http://ocw.mit.edu

6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 1

We're going to get started. Handouts are the by the door if anybody didn't pick one up. My name is Charles Leiserson. I will be lecturing this course this term, Introduction to Algorithms, with Erik Demaine. In addition, this is an SMA course, a Singapore MIT Alliance course which will be run in Singapore by David Hsu. And so all the lectures will be videotaped and made available on the Web for the Singapore students, as well as for MIT students who choose to watch them on the Web. If you have an issue of not wanting to be on the videotape, you should sit in the back row. OK? Otherwise, you will be on it.

There is a video recording policy, but it seems like they ran out. If anybody wants to see it, people, if they could just sort of pass them around maybe a little bit, once you're done reading it, or you can come up. I did secure one copy. Before we get into the content of the course, let's briefly go over the course information because there are some administrative things that we sort of have to do.

As you can see, this term we have a big staff. Take a look at the handout here. Including this term six TAs, which is two more TAs than we normally get for this course. That means recitations will be particularly small. There is a World Wide Web page, and you should bookmark that and go there regularly because that is where everything will be distributed. Email. You should not be emailing directly to, even though we give you our email addresses, to the individual members of the staff. You should email us generally. And the reason is you will get much faster response. And also, for any communications, generally we like to monitor what the communications are so it's helpful to have emails coming to everybody on the course staff. As I mentioned, we will be doing distance learning this term. And so you can watch lectures online if you choose to do that.

I would recommend, for people who have the opportunity to watch, to come live. It's better live. You get to interact. There's an intangible that comes with having it live. In fact, in addition to the videos, I meet weekly with the Singapore students so that they have a live session as well. Prerequisites. The prerequisites for this course are 6.042, which is Math for Computer Science, and 6.001. You basically need discrete mathematics and probability, as well as programming experience to take this course successfully. People do not have that background should not be in the class. We will be checking prerequisites. If you have any questions, please come to talk to us after class.

Let's see. Lectures are here. For SMA students, they have the videotapes and they will also have a weekly meeting. Students must attend a one-hour recitation session each week. There will be new material presented in the recitation. Unlike the lectures, they will not be online. Unlike the lectures, there will not be lecture notes distributed for the recitations in general. And, yet, there will be material there that is directly on the exams. And so every term we say oh, when did you cover that? That was in recitation. You missed that one. So, recitations are mandatory. And, in particular, also let me just mention your recitation instructor is the one who assigns

your final grade. So we have a grade meeting and keep everybody normal, but your recitation has the final say on your grade.

Handouts. Handouts are available on the course Web page. We will not generally, except for this one, first handout, be bringing handouts to class. Textbook is this book, Introduction to Algorithms. MIT students can get it any of the local bookstores, including the MIT Coop. There is also a new online service that provides textbooks. You can also get a discount if you buy it at the MIT Press Bookstore. There is a coupon in the MIT Student Telephone Directory for a discount on MIT Press books. And you can use that to purchase this book at a discount. Course website. This is the course website. It links to the Stellar website, which is where, actually, everything will be kept.

And SMA students have their own website. Some students find this course particularly challenges so we will have extra help. We will post weekly office hours on the course website for the TAs. And then as an experiment this term, we are going to offer homework labs for this class. What a homework lab is, is it's a place and a time you can go where other people in the course will go to do homework.

And there will be typically two TAs who staff the lab. And so, as you're working on your homework, you can get help from the TAs if you need it. And it's generally a place, we're going to schedule those, and they will be on the course calendar for where it is and when it is that they will be held, but usually Sundays 2:00 to 4:00 pm, or else it will be some evening. I think the first one is an evening, right?

Near to when the homework is due. Your best bet is try to do the homework in advance of the homework lab. But then, if you want extra help, if you want to talk over your solutions with people because as we will talk about problem sets you can solve in collaboration with other people in the class. In addition, there are several peer assistance programs. Also the office of Minority Education has an assistance program, and those usually get booked up pretty quickly. If you're interested in those, good idea to make an appointment to get there and get help soon. The homework labs, I hope a lot of people will try that out. We've never done this. I don't know of any other course. Do other people know of courses at MIT that have done this? 6.011 did it, OK.

Good. And was it successful in that class? It never went, OK. Good. [LAUGHTER] We will see. If it's not paying off then we will just return to ordinary office hours for those TAs, but I think for some students that is a good opportunity. If you wish to be registered in this course, you must sign up on the course Web page. So, that is requirement one. It must be done today. You will find it difficult to pass the course if you are not in the class. And you should notify your TA if you decide to drop so that we can get you off and stop the mailings, stop the spam. And you should register today before 7:00 PM.

And then we're going to email your recitation assignment to you before Noon tomorrow. And if you don't receive this information by Thursday Noon, please send us an email to the course staff generally, not to me individually, saying that you didn't receive your recitation assignment. And so if you haven't received it by Thursday Noon you want to. I think generally they are going to send them out tonight or at least by tomorrow morning.

Yeah. OK. SMA students don't have to worry about this. Problem sets. We have nine problem sets that we project will be assigned during the semester. A couple things about problem sets. Homeworks won't generally be accepted, if you have extenuating circumstances you should make prior arrangements with your recitation instructor. In fact, almost all of the administrative stuff, you shouldn't come to me to ask and say can I hand in something late? You should be talking to your recitation instructor.

You can read the other things about the form, but let me just mention that there are exercises that should be solved but not handed in as well to give you drill on the material. I highly recommend you doing the exercises. They both test your understanding of the material, and exercises have this way of finding themselves on quizzes. You're often asked to describe algorithms. And here is a little outline of what you can use to describe an algorithm. The grading policy is something that somehow I cover. And always every term there are at least a couple of students who pretend like I never showed them this. If you skip problems it has a nonlinear effect on your grade. Nonlinear, OK?

If you don't skip any problems, no effect on your grade. If you skip one problem, a hundredth of a letter grade, we can handle that. But two problems it's a tenth. And, as you see, by the time you have skipped like five letter grades, it is already five problems. This is not problem sets, by the way. This is problems, OK? You're down a third of a letter grade. And if you don't do nine or more, so that's typically about three to four problem sets, you don't pass the class. I always have some students coming at the end of the year saying oh, I didn't do any of my problems. Can you just pass me because I did OK on the exams? Answer no, a very simple answer because we've said it upfront. So, the problem sets are an integral part of the course. Collaboration policy.

This is extremely important so everybody pay attention. If you are asleep now wake up. Like that's going to wake anybody up, right? [LAUGHTER] The goal of homework. Professor Demaine and my philosophy is that the goal of homework is to help you learn the material. And one way of helping to learn is not to just be stuck and unable to solve something because then you're in no better shape when the exam roles around, which is where we're actually evaluating you.

So, you're encouraged to collaborate. But there are some commonsense things about collaboration. If you go and you collaborate to the extent that all you're doing is getting the information from somebody else, you're not learning the material and you're not going to do well on the exams. In our experience, students who collaborate generally do better than students who work alone. But you owe it to yourself, if you're going to work in a study group, to be prepared for your study group meeting. And specifically you should spend a half an hour to 45 minutes on each problem before you go to group so you're up to speed and you've tried out your ideas. And you may have solutions to some, you may be stuck on some other ones, but at least you applied yourself to it. After 30 to 45 minutes, if you cannot get the problem, just sitting there and banging your head against it makes no sense.

It's not a productive use of your time. And I know most of you have issues with having time on your hands, right? Like it's not there. So, don't go banging your head against problems that are too hard or where you don't understand what's going on or whatever. That's when the study group can help out. And, as I mentioned, we'll have

homework labs which will give you an opportunity to go and do that and coordinate with other students rather than necessarily having to form your own group.

And the TAs will be there. If your group is unable to solve the problem then talk to other groups or ask your recitation instruction. And, that's how you go about solving them. Writing up the problem sets, however, is your individual responsibility and should be done alone. You don't write up your problem solutions with other students, you write them up on your own. And you should on your problem sets, because this is an academic place, we understand that the source of academic information is very important, if you collaborated on solutions you should write a list of the collaborators. Say I worked with so and so on this solution. It does not affect your grade. It's just a question of being scholarly.

It is a violation of this policy to submit a problem solution that you cannot orally explain to a member of the course staff. You say oh, well, my write-up is similar to that other person's. I didn't copy them. We may ask you to orally explain your solution. If you are unable, according to this policy, the presumption is that you cheated. So, do not write up stuff that you don't understand. You should be able to write up the stuff that you understand. Understand why you're putting down what you're putting down. If it isn't obvious, no collaboration whatsoever is permitted on exams. Exams is when we evaluate you. And now we're not interested in evaluating other people, we're interested in evaluating you. So, no collaboration on exams. We will have a take-home exam for the second quiz.

You should look at the schedule. If there are problems with the schedule of that, we want to know early. And we will give you more details about the collaboration in the lecture on Monday, November 28th. Now, generally, the lectures here, they're mandatory and you have to know them, but I know that some people say gee, 9:30 is kind of early, especially on a Monday or whatever. It can be kind of early to get up.

However, on Monday, November 28th, you fail the exam if you do not show up to lecture on time. That one day you must show up. Any questions about that? That one day you must show up here, even if you've been watching them on the Web. And generally, if you think you have transgressed, the best is to come to us to talk about it. We can usually work something out. It's when we find somebody has transgressed from a third-party or from obvious analyses that we do with homeworks, that's when things get messy. So, if you think, for some reason or other, oh, I may have done something wrong, please come and talk to us. We actually were students once, too, albeit many years ago. Any questions? So, this class has great material.

Fabulous material. And it's really fun, but you do have to work hard. Let's talk content. This is the topic of the first part of the course. The first part of the course is focused on analysis. The second part of the course is focused on design. Before you can do design, you have to master a bunch of techniques for analyzing algorithms. And then you'll be in a position to design algorithms that you can analyze and that which are efficient. The analysis of algorithm is the theoretical study --

-- of computer program performance -- -- and resource usage. And a particular focus on performance. We're studying how to make things fast. In particular, computer programs. We also will discover and talk about other resources such as communication, such as memory, whether RAM memory or disk memory. There are

other resources that we may care about, but predominantly we focus on performance. Because this is a course about performance, I like to put things in perspective a little bit by starting out and asking, in programming, what is more important than performance?

If you're in an engineering situation and writing code, writing software, what's more important than performance? Correctness. Good. OK. What else? Simplicity can be. Very good. Yeah. Maintainability often much more important than performance. Cost. And what type of cost are you thinking? No, I mean cost of what? We're talking software here, right? What type of cost do you have in mind?

There are some costs that are involved when programming like programmer time. So, programmer time is another thing also that might be. Stability. Robustness of the software. Does it break all the time? What else? Come on. We've got a bunch of engineers here. A lot of things. How about features? Features can be more important. Having a wider collection of features than your competitors. Functionality. Modularity. Is it designed in a way where you can make changes in a local part of the code and you don't have to make changes across the code in order to affect a simple change in the functionality? There is one big one which definitely, especially in the `90s, was like the big thing in computers.

The big thing. Well, security actually. Good. I don't even have that one down. Security is excellent. That's actually been more in the 2000. Security has been far more important often than performance. Scalability has been important, although scalability, in some sense, is performance related. But, yes, scalability is good. What was the big breakthrough and why do people use Macintosh rather than Windows, those people who are of that religion?

User-friendliness. Wow. If you look at the number of cycles of computers that went into user-friendliness in the `90s, it grew from almost nothing to where it's now the vast part of the computation goes into user-friendly. So, all those things are more important than performance. This is a course on performance. Then you can say OK, well, why do we bother and why study algorithms and performance if it's at the bottom of the heap? Almost always people would rather have these other things than performance. You go off and you say to somebody, would I rather have performance or more user-friendliness?

It's almost always more important than performance. Why do we care then? Yeah? That wasn't user-friendly. Sometimes performance is correlated with user-friendliness, absolutely. Nothing is more frustrating than sitting there waiting, right? So, that's a good reason. What are some other reasons why? Sometimes they have real-time constraints so they don't actually work unless they perform adequately. Yeah? Hard to get, well, we don't usually quantify user-friendliness so I'm not sure, but I understand what you're saying. He said we don't get exponential performance improvements in user-friendliness.

We often don't get that in performance either, by the way. [LAUGHTER] Sometimes we do, but that's good. There are several reasons that I think are important. Once is that often performance measures the line between the feasible and the infeasible. We have heard some of these things. For example, when there are real-time requirements, if it's not fast enough it's simply not functional. Or, if it uses too much memory it's simply not going to work for you. And, as a consequence, what you find is algorithms are on the cutting edge of entrepreneurship. If you're talking about just

re-implementing stuff that people did ten years ago, performance isn't that important at some level. But, if you're talking about doing stuff that nobody has done before, one of the reasons often that they haven't done it is because it's too time-consuming.

Things don't scale and so forth. So, that's one reason, is the feasible versus infeasible. Another thing is that algorithms give you a language for talking about program behavior, and that turns out to be a language that has been pervasive through computer science, is that the theoretical language is what gets adopted by all the practitioners because it's a clean way of thinking about things. A good way I think about performance, and the reason it's on the bottom of the heap, is sort of like performance is like money, it's like currency. You say what good does a stack of hundred dollar bills do for you? Would you rather have food or water or shelter or whatever? And you're willing to pay those hundred dollar bills, if you have hundred dollar bills, for that commodity.

Even though water is far more important to your living. Well, similarly, performance is what you use to pay for user-friendliness. It's what you pay for security. And you hear people say, for example, that I want greater functionality, so people will program in Java, even though it's much slower than C, because they'll say it costs me maybe a factor of three or something in performance to program in Java.

But Java is worth it because it's got all these object-oriented features and so forth, exception mechanisms and so on. And so people are willing to pay a factor of three in performance. So, that's why you want performance because you can use it to pay for these other things that you want. And that's why, in some sense, it's on the bottom of the heap, because it's the universal thing that you quantify.

Do you want to spend a factor of two on this or spend a factor of three on security, et cetera? And, in addition, the lessons generalize to other resource measures like communication, like memory and so forth. And the last reason we study algorithm performance is it's tons of fun. Speed is always fun, right? Why do people drive fast cars, race horses, whatever? Rockets, et cetera, why do we do that? Because speed is fun. Ski. Who likes to ski? I love to ski. I like going fast on those skis. It's fun. Hockey, fast sports, right? We all like the fast sports. Not all of us, I mean. Some people say he's not talking to me. OK, let's move on. That's sort of a little bit of a notion as to why we study this, is that it does, in some sense, form a common basis for all these other things we care about.

And so we want to understand how can we generate money for ourselves in computation? We're going to start out with a very simple problem. It's one of the oldest problems that has been studied in algorithms, is the problem of sorting. We're going to actually study this for several lectures because sorting contains many algorithmic techniques. The sorting problem is the following. We have a sequence a_1, a_2 up to a_n of numbers as input. And our output is a permutation of those numbers.

A permutation is a rearrangement of the numbers. Every number appears exactly once in the rearrangement such that, I sometimes use a dollar sign to mean "such that," a_1 is less than or equal to a_2 prime. Such that they are monotonically increasing in size. Take a bunch of numbers, put them in order. Here's an algorithm to do it. It's called insertion sort. And we will write this algorithm in what we call

pseudocode. It's sort of a programming language, except it's got English in there often. And it's just a shorthand for writing for being precise.

So this sorts A from 1 to n. And here is the code for it. This is what we call pseudocode. And if you don't understand the pseudocode then you should ask questions about any of the notations. You will start to get used to it as we go on. One thing is that in the pseudocode we use indentation, where in most languages they have some kind of begin-end delimiters like curly braces or something in Java or C, for example.

We just use indentation. The whole idea of the pseudocode is to try to get the algorithms as short as possible while still understanding what the individual steps are. In practice, there actually have been languages that use indentation as a means of showing the nesting of things. It's generally a bad idea, because if things go over one page to another, for example, you cannot tell what level of nesting it is.

Whereas, with explicit braces it's much easier to tell. So, there are reasons why this is a bad notation if you were doing software engineering. But it's a good one for us because it just keeps things short and makes fewer things to write down. So, this is insertion sort. Let's try to figure out a little bit what this does. It basically takes an array A and at any point the thing to understand is, we're setting basically, we're running the outer loop from j is 2 to n, and the inner loop that starts at j minus 1 and then goes down until it's zero.

Basically, if we look at any point in the algorithm, we essentially are looking at some element here j. A of j, the jth element. And what we do essentially is we pull a value out here that we call the key. And at this point the important thing to understand, and we'll talk more about this in recitation on Friday, is that there is an invariant that is being maintained by this loop each time through.

And the invariant is that this part of the array is sorted. And the goal each time through the loop is to increase, is to add one to the length of the things that are sorted. And the way we do that is we pull out the key and we just copy values up like this. And keep copying up until we find the place where this key goes, and then we insert it in that place. And that's why it's called insertion sort. We just sort of move the things, copy the things up until we find where it goes, and then we put it into place. And now we have it from A from one to j is sorted, and now we can work on j plus one.

Let's give an example of that. Imagine we are doing 8, 2, 4, 9, 3, 6. We start out with j equals 2. And we figure out that we want to insert it there. Now we have 2, 8, 4, 9, 3, 6. Then we look at the four and say oh, well, that goes over here. We get 2, 4, 8, 9, 3, 6 after the second iteration of the outer loop. Then we look at 9 and discover immediately it just goes right there. Very little work to do on that step. So, we have exactly the same output after that iteration. Then we look at the 3 and that's going to be inserted over there.

2, 3, 4, 8, 9, 6. And finally we look at the 6 and that goes in there. 2, 3, 4, 6, 8, 9. And at that point we are done. Question? The array initially starts at one, yes. A[1...n], OK? So, this is the insertion sort algorithm. And it's the first algorithm that we're going to analyze. And we're going to pull out some tools that we have from our math background to help to analyze it. First of all, let's take a look at the issue of running time.

The running time depends, of this algorithm depends on a lot of things. One thing it depends on is the input itself. For example, if the input is already sorted -- -- then insertion sort has very little work to do. Because every time through it's going to be like this case. It doesn't have to shuffle too many guys over because they're already in place. Whereas, in some sense, what's the worst case for insertion sort? If it is reverse sorted then it's going to have to do a lot of work because it's going to have to shuffle everything over on each step of the outer loop.

In addition to the actual input it depends, of course, on the input size. Here, for example, we did six elements. It's going to take longer if we, for example, do six times ten to the ninth elements. If we were sorting a lot more stuff, it's going to take us a lot longer. Typically, the way we handle that is we are going to parameterize things in the input size. We are going to talk about time as a function of the size of things that we are sorting so we can look at what is the behavior of that. And the last thing I want to say about running time is generally we want upper bonds on the running time.

We want to know that the time is no more than a certain amount. And the reason is because that represents a guarantee to the user. If I say it's not going to run, for example, if I tell you here's a program and it won't run more than three seconds, that gives you real information about how you could use it, for example, in a real-time setting. Whereas, if I said here's a program and it goes at least three seconds, you don't know if it's going to go for three years. It doesn't give you that much guarantee if you are a user of it. Generally we want upper bonds because it represents a guarantee to the user.

There are different kinds of analyses that people do. The one we're mostly going to focus on is what's called worst-case analysis. And this is what we do usually where we define T of n to be the maximum time on any input of size n. So, it's the maximum input, the maximum it could possibly cost us on an input of size n. What that does is, if you look at the fact that sometimes the inputs are better and sometimes they're worse, we're looking at the worst case of those because that's the way we're going to be able to make a guarantee.

It always does something rather than just sometimes does something. So, we're looking at the maximum. Notice that if I didn't have maximum then T(n) in some sense is a relation, not a function, because the time on an input of size n depends on which input of size n. I could have many different times, but by putting the maximum at it, it turns that relation into a function because there's only one maximum time that it will take.

Sometimes we will talk about average case. Sometimes we will do this. Here T of n is then the expected time over all inputs of size n. It's the expected time. Now, if I talk about expected time, what else do I need to say here? What does that mean, expected time? I'm sorry. Raise your hand. Expected inputs. What does that mean, expected inputs? I need more math. What do I need by expected time here, math? You have to take the time of every input and then average them, OK. That's kind of what we mean by expected time. Good. Not quite. I mean, what you say is completely correct, except is not quite enough.

Yeah? It's the time of every input times the probability that it will be that input. It's a way of taking a weighted average, exactly right. How do I know what the probability

of every input is? How do I know what the probability a particular input occurs is in a given situation? I don't. I have to make an assumption. What's that assumption called? What kind of assumption do I have to meet? I need an assumption --

-- of the statistical distribution of inputs. Otherwise, expected time doesn't mean anything because I don't know what the probability of something is. In order to do probability, you need some assumptions and you've got to state those assumptions clearly. One of the most common assumptions is that all inputs are equally likely. That's called the uniform distribution. Every input of size n is equally likely, that kind of thing. But there are other ways that you could make that assumption, and they may not all be true. This is much more complicated, as you can see. Fortunately, all of you have a strong probability background. And so we will not have any trouble addressing these probabilistic issues of dealing with expectations and such.

If you don't, time to go and say gee, maybe I should take that Probability class that is a prerequisite for this class. The last one I am going to mention is best-case analysis. And this I claim is bogus. Bogus. No good. Why is best-case analysis bogus? Yeah? The best-case probably doesn't ever happen. Actually, it's interesting because for the sorting problem, the most common things that get sorted are things that are already sorted interestingly, or at least almost sorted. For example, one of the most common things that are sorted is check numbers by banks. They tend to come in, in the same order that they are written.

They're sorting things that are almost always sorted. I mean, it's good. When upper bond, not lower bound? Yeah, you want to make a guarantee. And so why is this not a guarantee? You're onto something there, but we need a little more precision here. How can I cheat? Yeah? Yeah, you can cheat. You cheat. You take any slow algorithm that you want and just check for some particular input, and if it's that input, then you say immediately yeah, OK, here is the answer. And then it's got a good best-case.

But I didn't tell you anything about the vast majority of what is going on. So, you can cheat with a slow algorithm that works fast on some input. It doesn't really do much for you so we normally don't worry about that. Let's see. What is insertion sorts worst-case time? Now we get into some sort of funny issues. First of all, it sort of depends on the computer you're running on. Whose computer, right? Is it a big supercomputer or is it your wristwatch? They have different computational abilities.

And when we compare algorithms, we compare them typically for relative speed. This is if you compared two algorithms on the same machine. You could argue, well, it doesn't really matter what the machine is because I will just look at their relative speed. But, of course, I may also be interested in absolute speed. Is one algorithm actually better no matter what machine it's run on? And so this kind of gets sort of confusing as to how I can talk about the worst-case time of an algorithm of a piece of software when I am not talking about the hardware because, clearly, if I had run on a faster machine, my algorithms are going to go faster. So, this is where you get the big idea of algorithms.

Which is why algorithm is such a huge field, why it spawns companies like Google, like Akamai, like Amazon. Why algorithmic analysis, throughout the history of computing, has been such a huge success, is our ability to master and to be able to take what is apparently a really messy, complicated situation and reduce it to being able to do some mathematics. And that idea is called asymptotic analysis.

And the basic idea of asymptotic analysis is to ignore machine-dependent constants -- -- and, instead of the actual running time, look at the growth of the running time. So, we don't look at the actual running time. We look at the growth. Let's see what we mean by that. This is a huge idea. It's not a hard idea, otherwise I wouldn't be able to teach it in the first lecture, but it's a huge idea. We are going to spend a couple of lectures understanding the implications of that and will basically be doing it throughout the term.

And if you go on to be practicing engineers, you will be doing it all the time. In order to do that, we adopt some notations that are going to help us. In particular, we will adopt asymptotic notation. Most of you have seen some kind of asymptotic notation. Maybe a few of you haven't, but mostly you should have seen a little bit. The one we're going to be using in this class predominantly is theta notation.

And theta notation is pretty easy notation to master because all you do is, from a formula, just drop low order terms and ignore leading constants. For example, if I have a formula like $3n^3 = 90n^2 - 5n + 6046$, I say, well, what low-order terms do I drop? Well, n^3 is a bigger term n^2 than. I am going to drop all these terms and ignore the leading constant, so I say that's $\Theta(n^3)$. That's pretty easy. So, that's theta notation. Now, this is an engineering way of manipulating theta notation. There is actually a mathematical definition for this, which we are going to talk about next time, which is a definition in terms of sets of functions. And, you are going to be responsible, this is both a math and a computer science engineering class.

Throughout the course you are going to be responsible both for mathematical rigor as if it were a math course and engineering commonsense because it's an engineering course. We are going to be doing both. This is the engineering way of understanding what you do, so you're responsible for being able to do these manipulations. You're also going to be responsible for understanding the mathematical definition of theta notion and of its related O notation and omega notation.

If I take a look as n approached infinity, a $\Theta(n^2)$ algorithm always beats, eventually, a $\Theta(n^3)$ algorithm. As n gets bigger, it doesn't matter what these other terms were if I were describing the absolute precise behavior in terms of a formula. If I had a $\Theta(n^2)$ algorithm, it would always be faster for sufficiently large n than a $\Theta(n^3)$ algorithm. It wouldn't matter what those low-order terms were. It wouldn't matter what the leading constant was. This one will always be faster.

Even if you ran the $\Theta(n^2)$ algorithm on a slow computer and the $\Theta(n^3)$ algorithm on a fast computer. The great thing about asymptotic notation is it satisfies our issue of being able to compare both relative and absolute speed, because we are able to do this no matter what the computer platform. On different platforms we may get different constants here, machine-dependent constants for the actual running time, but if I look at the growth as the size of the input gets larger, the asymptotics generally won't change. For example, I will just draw that as a picture. If I have n on this axis and T(n) on this axis.

This may be, for example, a $\Theta(n^3)$ algorithm and this may be a $\Theta(n^2)$ algorithm. There is always going to be some point n_o where for everything larger the $\Theta(n^2)$ algorithm is going to be cheaper than the $\Theta(n^3)$ algorithm not

matter how much advantage you give it at the beginning in terms of the speed of the computer you are running on. Now, from an engineering point of view, there are some issues we have to deal with because sometimes it could be that that n_o is so large that the computers aren't big enough to run the problem. That's why we, nevertheless, are interested in some of the slower algorithms, because some of the slower algorithms, even though they may not asymptotically be slower, I mean asymptotically they will be slower.

They may still be faster on reasonable sizes of things. And so we have to both balance our mathematical understanding with our engineering commonsense in order to do good programming. So, just having done analysis of algorithms doesn't automatically make you a good programmer. You also need to learn how to program and use these tools in practice to understand when they are relevant and when they are not relevant. There is a saying.

If you want to be a good program, you just program ever day for two years, you will be an excellent programmer. If you want to be a world-class programmer, you can program every day for ten years, or you can program every day for two years and take an algorithms class. Let's get back to what we were doing, which is analyzing insertion sort. We are going to look at the worse-case. Which, as we mentioned before, is when the input is reverse sorted. The biggest element comes first and the smallest last because now every time you do the insertion you've got to shuffle everything over.

You can write down the running time by looking at the nesting of loops. What we do is we sum up. What we assume is that every operation, every elemental operation is going to take some constant amount of time. But we don't have to worry about what that constant is because we're going to be doing asymptotic analysis. As I say, the beautify of the method is that it causes all these things that are real distinctions to sort of vanish.

We sort of look at them from 30,000 feet rather than from three millimeters or something. Each of these operations is going to sort of be a basic operation. One way to think about this, in terms of counting operations, is counting memory references. How many times do you actually access some variable? That's another way of sort of thinking about this model. When we do that, well, we're going to go through this loop, j is going from 2 to n, and then we're going to add up the work that we do within the loop. We can sort of write that in math as summation of j equals 2 to n. And then what is the work that is going on in this loop? Well, the work that is going on in this loop varies, but in the worst case how many operations are going on here for each value of j?

For a given value of j, how much work goes on in this loop? Can somebody tell me? Asymptotically. It's j times some constant, so it's theta j. So, there is theta j work going on here because this loop starts out with i being j minus 1, and then it's doing just a constant amount of stuff for each step of the value of i, and i is running from j minus one down to zero. So, we can say that is theta j work that is going on. Do people follow that?

OK. And now we have a formula we can evaluate. What is the evaluation? If I want to simplify this formula, what is that equal to? Sorry. In the back there. Yeah. OK. That's just Theta(n^2), good. Because when you're saying is the sum of consecutive numbers, you mean what? What's the mathematic term we have for that so we can

communicate? You've got to know these things so you can communicate. It's called what type of sequence? It's actually a series, but that's OK. What type of series is this called? Arithmetic series, good. Wow, we've got some sharp people who can communicate.

This is an arithmetic series. You're basically summing 1 + 2 + 3 + 4, some constants in there, but basically it's 1 + 2 + 3 + 4 + 5 + 6 up to n. That's Theta(n^2). If you don't know this math, there is a chapter in the book, or you could have taken the prerequisite. Erythematic series. People have this vague recollection. Oh, yeah. Good. Now, you have to learn these manipulations. We will talk about a bit next time, but you have to learn your theta manipulations for what works with theta. And you have to be very careful because theta is a weak notation. A strong notation is something like Leibniz notation from calculus where the chain rule is just canceling two things.

It's just fabulous that you can cancel in the chain rule. And Leibniz notation just expresses that so directly you can manipulate. Theta notation is not like that. If you think it is like that you are in trouble. You really have to think of what is going on under the theta notation. And it is more of a descriptive notation than it is a manipulative notation. There are manipulations you can do with it, but unless you understand what is really going on under the theta notation you will find yourself in trouble.

And next time we will talk a little bit more about theta notation. Is insertion sort fast? Well, it turns out for small n it is moderately fast. But it is not at all for large n. So, I am going to give you an algorithm that is faster. It's called merge sort. I wonder if I should leave insertion sort up. Why not. I am going to write on this later, so if you are taking notes, leave some space on the left. Here is merge sort of an array A from 1 up to n.

And it is basically three steps. If n equals 1 we are done. Sorting one element, it is already sorted. All right. Recursive algorithm. Otherwise, what we do is we recursively sort A from 1 up to the ceiling of n over 2. And the array A of the ceiling of n over 2 plus one up to n. So, we sort two halves of the input. And then, three, we take those two lists that we have done and we merge them.

And, to do that, we use a merge subroutine which I will show you. The key subroutine here is merge, and it works like this. I have two lists. Let's say one of them is 20. I am doing this in reverse order. I have sorted this like this. And then I sort another one. I don't know why I do it this order, but anyway. Here is my other list. I have my two lists that I have sorted. So, this is A[1] to A[|n/2|] and A[|n/2|+1] to A[n] for the way it will be called in this program. And now to merge these two, what I want to do is produce a sorted list out of both of them.

What I do is first observe where is the smallest element of any two lists that are already sorted? It's in one of two places, the head of the first list or the head of the second list. I look at those two elements and say which one is smaller? This one is smaller. Then what I do is output into my output array the smaller of the two. And I cross it off. And now where is the next smallest element? And the answer is it's going to be the head of one of these two lists. Then I cross out this guy and put him here and circle this one. Now I look at these two guys. This one is smaller so I output that and circle that one. Now I look at these two guys, output 9. So, every step here is

some fixed number of operations that is independent of the size of the arrays at each step.

Each individual step is just me looking at two elements and picking out the smallest and advancing some pointers into the array so that I know where the current head of that list is. And so, therefore, the time is order n on n total elements. The time to actually go through this and merge two lists is order n. We sometimes call this linear time because it's not quadratic or whatever. It is proportional to n, proportional to the input size. It's linear time. I go through and just do this simple operation, just working up these lists, and in the end I have done essentially n operations, order n operations each of which cost constant time.

That's a total of order n time. Everybody with me? OK. So, this is a recursive program. We can actually now write what is called a recurrence for this program. The way we do that is say let's let the time to sort n elements to be T(n). Then how long does it take to do step one? That's just constant time. We just check to see if n is 1, and if it is we return. That's independent of the size of anything that we are doing. It just takes a certain number of machine instructions on whatever machine and we say it is constant time. We call that theta one. This is actually a little bit of an abuse if you get into it.

And the reason is because typically in order to say it you need to say what it is growing with. Nevertheless, we use this as an abuse of the notation just to mean it is a constant. So, that's an abuse just so people know. But it simplifies things if I can just write theta one. And it basically means the same thing. Now we recursively sort these two things. How can I describe that? The time to do this, I can describe recursively as T of ceiling of n over 2 plus T of n minus ceiling of n over 2. That is actually kind of messy, so what we will do is just be sloppy and write 2T(n/2). So, this is just us being sloppy.

And we will see on Friday in recitation that it is OK to be sloppy. That's the great thing about algorithms. As long as you are rigorous and precise, you can be as sloppy as you want. [LAUGHTER] This is sloppy because I didn't worry about what was going on, because it turns out it doesn't make any difference. And we are going to actually see that that is the case. And, finally, I have to merge the two sorted lists which have a total of n elements. And we just analyze that using the merge subroutine. And that takes us to theta n time.

That allows us now to write a recurrence for the performance of merge sort. Which is to say that T of n is equal to theta 1 if n equals 1 and 2T of n over 2 plus theta of n if n is bigger than 1. Because either I am doing step one or I am doing all steps one, two and three. Here I am doing step one and I return and I am done. Or else I am doing step one, I don't return, and then I also do steps two and three. So, I add those together. I could say theta n plus theta 1, but theta n plus theta 1 is just theta n because theta 1 is a lower order term than theta n and I can throw it away. It is either theta 1 or it is 2T of n over 2 plus theta n. Now, typically we won't be writing this.

Usually we omit this. If it makes no difference to the solution of the recurrence, we will usually omit constant base cases. In algorithms, it's not true generally in mathematics, but in algorithms if you are running something on a constant size input it takes constant time always. So, we don't worry about what this value is. And it turns out it has no real impact on the asymptotic solution of the recurrence.

How do we solve a recurrence like this? I now have T of n expressed in terms of T of n over 2. That's in the book and it is also in Lecture 2. We are going to do Lecture 2 to solve that, but in the meantime what I am going to do is give you a visual way of understanding what this costs, which is one of the techniques we will elaborate on next time. It is called a recursion tree technique. And I will use it for the actual recurrence that is almost the same 2T(n/2), but I am going to actually explicitly, because I want you to see where it occurs, plus some constant times n where c is a constant greater than zero. So, we are going to look at this recurrence with a base case of order one.

I am just making the constant in here, the upper bound on the constant be explicit rather than implicit. And the way you do a recursion tree is the following. You start out by writing down the left-hand side of the recurrence. And then what you do is you say well, that is equal to, and now let's write it as a tree. I do c of n work plus now I am going to have to do work on each of my two children.

T of n over 2 and T of n over 2. If I sum up what is in here, I get this because that is what the recurrence says, T(n)=2T(n/2)+cn. I have 2T(n/2)+cn. Then I do it again. I have cn here. I now have here cn/2. And here is cn/2. And each of these now has a T(n/4). And these each have a T(n/4). And this has a T(n/4). And I keep doing that, the dangerous dot, dot, dots. And, if I keep doing that, I end up with it looking like this.

And I keep going down until I get to a leaf. And a leaf, I have essentially a T(1). That is T(1). And so the first question I ask here is, what is the height of this tree? Yeah. It's log n. It's actually very close to exactly log n because I am starting out at the top with n and then I go to n/2 and n/4 and all the way down until I get to 1. The number of halvings of n until I get to 1 is log n so the height here is log n. It's OK if it is constant times log n. It doesn't matter. How many leaves are in this tree, by the way?

How many leaves does this tree have? Yeah. The number of leaves, once again, is actually pretty close. It's actually n. If you took it all the way down. Let's make some simplifying assumption. n is a perfect power of 2, so it is an integer power of 2. Then this is exactly log n to get down to T(1). And then there are exactly n leaves, because the number of leaves here, the number of nodes at this level is 1, 2, 4, 8.

And if I go down height h, I have 2 to the h leaves, 2 to the log n, that is just n. We are doing math here, right? Now let's figure out how much work, if I look at adding up everything in this tree I am going to get T(n), so let's add that up. Well, let's add it up level by level. How much do we have in the first level? Just cn. If I add up the second level, how much do I have? cn. How about if I add up the third level? cn. How about if I add up all the leaves? Theta n.

It is not necessarily cn because the boundary case may have a different constant. It is actually theta n, but cn all the way here. If I add up the total amount, that is equal to cn times log n, because that's the height, that is how many cn's I have here, plus theta n. And this is a higher order term than this, so this goes away, get rid of the constants, that is equal to theta(n lg n). And theta(n lg n) is asymptotically faster than theta(n^2). So, merge sort, on a large enough input size, is going to beat insertion sort.

Merge sort is going to be a faster algorithm. Sorry, you guys, I didn't realize you couldn't see over there. You should speak up if you cannot see. So, this is a faster algorithm because theta(n lg n) grows more slowly than theta(n^2). And merge sort asymptotically beats insertion sort. Even if you ran insertion sort on a supercomputer, somebody running on a PC with merge sort for sufficient large input will clobber them because actually n^2 is way bigger than n log n once you get the n's to be large. And, in practice, merge sort tends to win here for n bigger than, say, 30 or so.

If you have a very small input like 30 elements, insertion sort is a perfectly decent sort to use. But merge sort is going to be a lot faster even for something that is only a few dozen elements. It is going to actually be a faster algorithm. That's sort of the lessons, OK? Remember that to get your recitation assignments and attend recitation on Friday. Because we are going to be going through a bunch of the things that I have left on the table here. And see you next Monday.