

[www.mientayvn.com](http://www.mientayvn.com)

Đây là bản ghi lại lời giảng của giáo sư Erik Demaine and Charles Leiserson trên lớp. Xem toàn bộ bài giảng này bạn có thể tìm <http://www.mientayvn.com> > Học li u m > Học vi n công ngh Massachusetts > Toán h c > Nh p môn gi i thu t > ch ã ng I.

MIT OpenCourseWare

<http://ocw.mit.edu>

6.046J Introduction to Algorithms, Fall 2005

Please use the following citation format:

Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms, Fall 2005*. (Massachusetts Institute of Technology: MIT OpenCourseWare). <http://ocw.mit.edu> (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>

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 –

Chúng ta hãy nói về nội dung. Đây là chủ đề của phần đầu của khóa học. Phần đầu của khóa học tập trung vào việc phân tích. Phần thứ hai của khóa học tập trung vào việc thiết kế. Trước khi bạn có thể thiết kế, bạn phải nắm vững một loạt các kỹ thuật để phân tích giải thuật. Và sau đó bạn sẽ ở vị trí thiết kế giải thuật lúc đó bạn cần phải phân tích giải thuật nào có hiệu quả hơn. Phân tích giải thuật là nghiên cứu lí thuyết--

-- 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?

--của hiệu suất chương trình máy tính - - và cách sử dụng nguồn tài nguyên. Và tập trung đặc biệt vào hiệu suất. Chúng ta sẽ học cách để làm cho chúng chạy nhanh. Nói riêng, chương trình máy tính. Chúng ta cũng sẽ khám phá và nói về những tài nguyên khác như truyền thông, như bộ nhớ, bất kể bộ nhớ RAM hay bộ nhớ đĩa. Có những tài nguyên khác mà chúng ta có thể quan tâm, nhưng chủ yếu chúng ta tập trung vào hiệu suất. Bởi vì đây là khóa học về hiệu suất, tôi muốn đặt mọi thứ đúng bối cảnh một chút bằng cách khởi đầu và hỏi, trong lập trình, yếu tố nào là quan trọng hơn hiệu suất?

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?

Nếu bạn ở vị trí kĩ sư hoặc viết mã, viết phần mềm, điều gì quan trọng hơn hiệu suất? Sự chính xác. Tốt. Vâng. Cái gì nữa? Có thể là sự đơn giản. Rất tốt. Vâng. Có thể điều chỉnh được thường quan trọng hơn nhiều so với hiệu suất. Giá thành. Và loại giá nào mà bạn sẽ nghĩ tới? Không, tôi muốn nói giá của cái gì? Chúng ta đang nói về phần mềm ở đây, đúng không? Loại giá nào mà bạn có trong đầu.

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.

Có một số giá thành phải được gộp vào khi lập trình như là thời gian lập trình. Vậy, thời gian lập trình cũng là một thứ cần phải kể. Độ ổn định. Sức chịu đựng của phần mềm. Nó có bị lỗi theo thời gian không? Còn gì nữa? Thôi nào. Chúng ta có một lũ kĩ sư ở đây. Nhiều thứ. Còn về tính năng thì sao? Tính năng có thể quan trọng hơn. Phải có tập hợp các tính năng nhiều hơn đối thủ của bạn. Chức năng. Tính có modun. Nó được thiết kế sao cho bạn có thể thay đổi từng phần của mã mà không thay đổi toàn bộ mã để đạt được một sự thay đổi nhỏ trong chức năng? Có một vấn đề lớn mà rõ ràng, thuộc loại vấn đề lớn trong máy tính, đặc biệt vào những năm 90.

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?

Vấn đề lớn. Vâng, vấn đề bảo mật. Tốt. Thậm chí tôi không dám nói tôi nắm vững điều đó. Bảo mật thật xuất sắc. Điều đó thật sự cần hơn vào năm 2000. Bảo mật quan trọng hơn nhiều so với hiệu suất. Khả năng mở rộng cũng quan trọng, mặc dù khả năng mở rộng, và theo nghĩa nào đó, có quan hệ với hiệu suất. Nhưng, vâng, khả năng mở rộng là tốt. Cái gì là bước nhảy vọt lớn và tại sao người ta dùng Macintosh nhiều hơn Windows, những người này thuộc về tôn giáo đó phải không?

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?

Sự thân thiện với người dùng. Wow. Nếu bạn xét số chu kì máy tính ở khía cạnh thân thiện với người dùng vào những năm 90, nó phát triển từ chỗ hầu như không có gì đến chỗ bây giờ phần lớn các thao tác điện toán đều thân thiện với người dùng. Vì vậy, tất cả những thứ này quan trọng hơn hiệu suất. Đây là khóa học về hiệu suất. Do đó bạn có thể nói OK, vâng, tại sao chúng ta bị quấy rầy và tại sao nghiên cứu thuật giải và hiệu suất nếu nó ở dưới đồng này? Hầu như luôn luôn người ta muốn những thứ này chứ không phải hiệu suất. Bạn đi ra và bạn nói với ai đó, tôi nên ưu tiên cho hiệu suất hơn hay là thân thiện với người dùng hơn?

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.

Nó hầu như luôn luôn quan trọng hơn hiệu suất. Do đó tại sao chúng ta quan tâm đến hiệu suất? Vâng? Mà không phải là sự thân thiện với người dùng. Đôi khi hiệu suất có sự tương quan với sự thân thiện người dùng, chính xác. Không có gì làm ta bực mình hơn ngồi chờ, đúng không? Vì vậy, đó là lí do tốt. Một số lí do khác là gì tại sao? Thành thạo chúng có sự ràng buộc thời gian thực vì vậy chúng không thực sự hoạt động nếu chúng không thực hiện thỏa đáng. Vâng? Khó để biết, vâng, thường khó để định lượng sự thân thiện với người dùng vì vậy tôi không chắc nhưng tôi hiểu những gì bạn nói. Anh ta nói chúng ta không thể vừa nhận được sự cải tiến hiệu suất theo luật hàm mũ và sự thân thiện với người dùng cùng một lúc.

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. One 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.

Nhân tiện, chúng ta thường không nhận được điều đó trong hiệu suất. Thành thạo chúng ta làm, nhưng điều đó tốt. Có một vài lí do mà tôi nghĩ là quan trọng. Thứ nhất là hiệu suất thường đo ranh giới giữa sự thích đáng và không thích đáng. Chúng ta đã nghe một số trong những thứ này. Chẳng hạn, khi có một yêu cầu thời gian thực, nếu nó không đủ nhanh, đơn giản là nó sẽ không làm việc. Hoặc nếu nó dùng quá nhiều bộ nhớ, nó đơn giản sẽ không làm việc cho bạn. Và, vì vậy, những gì chúng ta tìm được trong giải thuật vượt xa tinh thần doanh nhân. Nếu bạn nói về một món thực thi lại mà người ta đã làm 10 năm trước, hiệu suất không quan trọng ở mức nào đó. Nhưng, nếu bạn nói về việc làm món mà không ai làm trước đây, một trong những lí do mà họ đã không làm nó bởi vì nó tiêu tốn quá nhiều thời gian.

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.

Những thứ này không so sánh được và vâng vâng. Vì vậy, đó là một lí do, là thích đáng với không thích đáng. Một thứ khác là giải thuật cho bạn một ngôn ngữ để nói về sự diễn biến chương trình, và hóa ra là một ngôn ngữ đã lan truyền khắp nơi trong khoa học máy tính, là ngôn ngữ lí thuyết là những gì đã được thông qua bởi tất cả những người đang hành nghề vì nó là cách nghĩ sáng sủa về các thứ. Một cách tốt mà tôi nghĩ về hiệu suất, và lí do nó ở dưới các tiêu chí khác, là phần nào hiệu suất nào tiền nấy, khi bạn có một sắp tiền một trăm đô la bạn sẽ mua gì? Bạn thích cái nào, thực phẩm hoặc nước hoặc chỗ ở hay bất cứ thứ gì đó? Và bạn vui lòng trả những tờ một trăm đô la này cho tiện nghi đó.

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.

Cho dù nước quan trọng hơn nhiều đối với cuộc sống của bạn. Vâng, tương tự, hiệu suất là những gì bạn dùng để trả giá cho sự thân thiện với người dùng. Nó là những gì bạn phải trả giá cho sự bảo mật. Và bạn nghe người ta nói, chẳng hạn, rằng tôi muốn nhiều chức năng hơn, vì vậy người ta sẽ lập trình bằng Java, cho dù nó chậm hơn nhiều so với C, bởi vì họ nói khi lập trình bằng Java hiệu suất sẽ giảm đi gần ba lần.

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.

Nhưng Java đáng để bỏ thời gian vì nó có tất cả những tính năng hướng đối tượng và v.v., những cơ chế loại trừ và v.v.. Và vì vậy người ta sẽ vui lòng trả giá cho sự giảm hiệu suất gấp 3 lần. Vì vậy, đó là lí do tại sao bạn muốn hiệu suất vì bạn có thể dùng nó để thanh toán cho những thứ khác ở đây mà bạn muốn. Và đó là lí do tại sao, theo một quan điểm nào đó, nó ở dưới của các tiêu chí khác, vì nó là thứ phổ biến mà bạn định lượng.

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.

Bạn có muốn tốn 2 lần trên cái này hoặc tốn gấp 3 lần cho bảo mật, v.v..? Và, thêm vào đó, những điều này có thể tổng quát hóa cho sự đo các tài nguyên khác như truyền thông, bộ nhớ và v.v...Và lí do cuối cùng chúng tôi nghiên cứu hiệu suất thuật toán là nó rất vui. Tốc độ luôn luôn vui, đúng không? Tại sao người ta lái xe nhanh, đua ngựa, hay bất cứ thứ gì khác? Tên lửa, v.v., tại sao chúng ta làm điều đó? Bởi vì tốc độ vui. Trượt tuyết. Ai thích trượt tuyết? Tôi thích trượt tuyết. Tôi thích lướt nhanh trên những ván trượt này. Nó vui. Môn khúc côn cầu, những môn thể thao nhanh, đúng không? Tất cả chúng ta đều thích những môn thể thao nhanh. Ý tôi là, không phải tất cả chúng ta. Một số người nói rằng thầy đừng nói về tôi. Vâng, chúng ta hãy tiếp tục. Đó là một ít ý niệm về việc tại sao chúng ta nghiên cứu cái này, là vì theo một quan điểm nào đó, nó hình thành nên cơ sở chung cho tất cả những thứ mà chúng ta quan tâm này.

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.

Và vì vậy chúng ta muốn tìm hiểu cách mà chúng ta có thể tạo ra tiền cho chính mình trong tính toán? Chúng ta sẽ khởi đầu với một vấn đề đơn giản. Nó là một trong những vấn đề cổ nhất đã từng được nghiên cứu trong giải thuật, là vấn đề sắp xếp. Chúng ta sẽ thực sự học về vấn đề này trong vài bài giảng vì sắp xếp chứa nhiều kĩ thuật giải thuật. Bài toán sắp xếp là như sau. Chúng ta có một dãy các số đầu vào  $a_1, a_2$  tới  $a_n$ . Và đầu ra của chúng ta là hoán vị của những số này.

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.

Hoán vị là sự sắp xếp lại các số. Mỗi số xuất hiện đúng một lần trong sự sắp xếp lại sao cho, tôi thỉnh thoảng dùng dấu đô la để kí hiệu cho "sao cho"  $a_1$  nhỏ hơn hoặc bằng  $a_2$  thấy. Sao cho chúng tăng dần đơn điệu về độ lớn. Lấy một chuỗi số, đặt chúng theo thứ tự. Đây là một giải thuật để làm. Nó được gọi là chèn trực tiếp. Và chúng ta sẽ viết giải thuật này dưới dạng giả mã. Nó là phần nào của ngôn ngữ lập trình, ngoại trừ thường có tiếng Anh trong đó. Và nó chỉ là sự viết tắt kí để viết chính xác.

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.

Vì vậy, đây là sự sắp xếp A từ 1 đến n. Và đây là mã cho nó. Đây là những gì mà chúng ta gọi là giả mã. Và nếu bạn không hiểu giả mã thì bạn sẽ hỏi về bất cứ chú thích nào. Bạn sẽ bắt đầu quen với nó khi chúng ta tiếp tục. Một điều là trong giả mã chúng ta dùng thụt đầu dòng, trong khi trong đa số các ngôn ngữ chúng ta có một vài loại dấu phân cách đầu-cuối như dấu ngoặc nhọn hoặc thứ gì đó trong Java hoặc C chẳng hạn.

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.

Chúng ta chỉ dùng thụt đầu dòng. Toàn bộ ý tưởng của giả mã là cố nhận được giải thuật ngắn nhất trong khi vẫn còn hiểu được từng bước là gì. Trong thực tế, đã có những ngôn ngữ dùng dấu thụt đầu dòng như một phương tiện để biểu diễn sự lồng vào nhau của các thứ. Nói chung, đó là một ý tưởng không hay, vì nếu chúng ta viết chương trình trên nhiều trang thì chúng ta không thể biết nó là cấp lồng gì.

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.

Ngược lại, với những dấu mốc rõ ràng nó dễ dàng hơn để nói. Vì vậy, đó là những lí do tại sao đây là ý tưởng không hay nếu bạn đang làm công nghệ phần mềm. Nhưng nó là một thứ tốt cho chúng ta vì nó làm cho mọi thứ ngắn và làm cho ít thứ được viết ra. Vì vậy, đây là chèn trực tiếp. Hãy thử tìm hiểu một chút xem cái này làm gì. Về cơ bản, nó lấy một mảng A và tại bất kì điểm nào điều cần hiểu là, về cơ bản chúng ta thiết lập, chúng ta sẽ chạy ở vòng bên ngoài từ j bằng 2 đến n, và vòng bên trong bắt đầu tại j-1 và sau đó giảm xuống cho đến khi nó bằng 0.

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.

Về cơ bản, nếu chúng ta nhìn vào bất cứ điểm nào trong giải thuật, về cơ bản chúng ta sẽ xét một yếu tố nào đó ở đây j. A (j), yếu tố thứ j. Và về cơ bản những gì chúng ta làm là chúng ta kéo một giá trị ra ngoài đây mà chúng ta gọi là khóa. Và tại điểm này điều quan trọng cần hiểu là, và chúng ta sẽ nói về điều này trong bài học vào ngày thứ sáu, là có một sự bất biến đã được giữ lại bởi vòng lặp này mỗi lần đi qua.

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.

Và sự bất biến là phần này của mảng được sắp xếp. Và mục tiêu mỗi lần qua vòng lặp là để tăng, là để cộng một vào độ dài của phần đã được sắp xếp. Và cách mà chúng ta làm điều đó là chúng ta kéo khóa ra và chúng ta copy giá trị lên giống như thế này. Và tiếp tục copy lên cho đến khi nào chúng ta tìm được nơi khóa này đến, và sau đó chúng ta chèn nó vào chỗ đó. Và đó là lí do tại sao nó được gọi là chèn trực tiếp. Chúng ta chỉ phần nào di chuyển các thứ, copy các thứ lên cho đến khi chúng ta tìm được nơi nó đến, và sau đó chúng ta đặt nó vào đúng chỗ. Và bây giờ chúng ta có nó A từ 1 đến j được sắp xếp, và bây giờ chúng ta có thể làm việc trên j cộng 1.

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.

Hãy xét một ví dụ về điều đó. Hãy tưởng tượng chúng ta sẽ sắp xếp 8, 2, 4, 9, 3, 6. Chúng ta khởi đầu với j bằng 2. Và chúng ta đoán ra rằng chúng ta muốn chèn nó ở đó. Bây giờ chúng ta được 2, 8, 4, 9, 3, 6. Sau đó chúng ta xét 4 và nói oh, vâng, cái đó đi lên đến đây. **Chúng ta có 2, 4, 8, 9, 3, 6** sau khi lặp lại lần hai vòng lặp ngoài. Sau đó chúng ta xét 9 và ngay lập tức khám phá ra nó chính xác đến ngay đó. Rất ít công việc để làm ở bước đó. Vì vậy chúng ta có chính xác đầu ra tương tự sau lần lặp lại đó. Rồi chúng ta xét 3 và cái đó sẽ được chèn vào đây kia.

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.

2, 3, 4, 8, 9, 6. Và cuối cùng chúng ta xét 6 và nó đi đến đó. . 2, 3, 4, 6, 8, 9. Và tại điểm này chúng ta đã hoàn thành. Có câu hỏi nào không? Mảng ban đầu bắt đầu tại một, vâng.  $A[1..n]$ , đúng không? Vì vậy, đây là thuật toán chèn trực tiếp. Và nó là thuật toán đầu tiên mà chúng ta sẽ phân tích. Và chúng ta sẽ kéo ra một số công cụ mà chúng ta có từ nền tảng toán học của chúng ta để giúp phân tích nó. Trước hết, chúng ta hãy xét vấn đề thời gian chạy.

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.

Thời gian chạy, của thuật toán này phụ thuộc vào nhiều thứ. Một thứ trong số đó chính là đầu vào. Chẳng hạn, nếu đầu vào được sắp xếp rồi - - thì chèn trực tiếp có ít công việc để làm. Bởi vì mỗi lần đi qua nó sẽ giống như trường hợp này. Nó không phải lê bước qua quá nhiều gã bởi vì chúng đã đứng chỗ rồi. Ngược lại, trên quan điểm nào đó, điều gì là xấu nhất cho sự chèn trực tiếp? Nếu nó bị sắp xếp ngược thì nó sẽ phải làm nhiều công việc bởi vì nó phải lê bước qua mọi thứ ở mỗi bước của vòng lặp ngoài.

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 bounds on the running time.

Tất nhiên là nó còn phụ thuộc kích thước đầu vào. Ở đây, chẳng hạn, chúng ta xét 6 yếu tố. Nó sẽ mất thời gian dài nếu chẳng hạn chúng ta thực hiện với 6 nhân 10 mũ 19 yếu tố. Nếu chúng ta phân loại nhiều nguyên liệu hơn, nó sẽ làm mất thời gian của chúng ta lâu hơn. Thông thường, cách mà chúng ta xử lý điều đó là chúng ta sẽ tham số hóa kích thước đầu vào. Chúng ta sẽ nói về thời gian như là một hàm của kích thước các thứ mà chúng ta sắp xếp vì vậy chúng ta có thể xét diễn biến của cái đó. Và điều cuối cùng mà tôi muốn nói về thời gian chạy là nói chung chúng ta muốn những ràng buộc cao hơn về thời gian chạy.

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 bounds because it represents a guarantee to the user.

Chúng ta muốn biết rằng thời gian không nhiều hơn một lượng nào đó. Và lý do là vì cái đó thể hiện một sự đảm bảo với người dùng. Nếu tôi nói nó sẽ không chạy, chẳng hạn, nếu tôi bảo bạn đây là chương trình và nó sẽ không chạy quá 3 giây, điều đó cho bạn một thông tin thực về cách bạn có thể dùng nó, chẳng hạn, trong thiết lập thời gian thực. Trong khi đó, nếu tôi nói đây là chương trình và nó chạy ít nhất sau 3 giây, bạn không biết rằng có khi nào sau ba năm nó mới chạy hay không. Nó không cho bạn nhiều sự đảm bảo nếu bạn là người dùng nó. Nói chung, chúng ta muốn có những ràng buộc cao hơn bởi vì nó biểu hiện một sự đảm bảo với người dùng.

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.

Có nhiều loại phép phân tích khác mà người ta dùng. Cái mà chúng ta chủ yếu sẽ tập trung vào là phân tích trường hợp xấu nhất. Và đây là những gì chúng ta làm, thông

thường ở đó chúng ta định nghĩa  $T(n)$  là thời gian cực đại trên bất cứ đầu vào kích thước  $n$  nào. Vì vậy, nó là đầu vào cực đại, cực đại mà nó đòi hỏi chúng ta trên đầu vào kích thước  $n$ . Những gì mà cái đó làm là, nếu bạn xét sự kiện là thỉnh thoảng đầu vào tốt hơn hoặc thỉnh thoảng chúng tệ hơn, chúng ta sẽ xét trường hợp xấu nhất của những cái này bởi vì đó là cách mà chúng ta có thể tạo ra sự bảo đảm.

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.

Luôn luôn làm thứ gì đó thay vì chỉ thỉnh thoảng làm. Vì vậy, chúng ta sẽ xét cực đại. Chú ý rằng nếu tôi không có cực đại thì  $T(n)$  theo một quan điểm nào đó là một hệ thức, không phải hàm, vì thời gian trên đầu vào kích thước  $n$  phụ thuộc vào đầu vào kích thước  $n$ . Tôi có thể có nhiều thời gian khác nhau, nhưng bằng cách đặt cực đại tại nó, nó chuyển hệ thức đó thành hàm vì chỉ có một thời gian cực đại mà nó sẽ lấy.

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.

Thỉnh thoảng chúng ta sẽ nói về trường hợp trung bình. Thỉnh thoảng chúng ta sẽ làm điều này. Do đó, ở đây  $T(n)$  là thời gian dự kiến trên tất cả đầu vào kích thước  $n$ . Nó là thời gian dự kiến. Bây giờ, nếu tôi nói về thời gian dự kiến, tôi cần phải nói ở đây cái gì nữa? Điều đó có nghĩa là gì, thời gian dự kiến? Tôi xin lỗi. Giơ tay lên nào. Các đầu vào dự kiến. Điều đó có nghĩa là gì, các đầu vào dự kiến? Tôi cần thêm toán học. Tôi cần gì vào khoảng thời gian dự kiến ở đây, toán học? Bạn phải lấy thời gian của mỗi đầu vào và sau đó tính trung bình chúng, OK. Đó phần nào là những gì mà chúng ta muốn nói qua thời gian dự kiến. Tốt. Không hoàn toàn. Ý tôi là, những gì bạn nói là hoàn toàn chính xác, ngoại trừ là chưa đủ.

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 – Đúng không? Nó là thời gian của mỗi đầu vào nhân với xác suất mà nó sẽ ở đầu vào đó. Đó là cách lấy trung bình có trọng số, đúng chính xác. Làm sao mà tôi biết xác suất ở mỗi đầu vào là gì? Làm sao mà tôi biết xác suất mà một đầu vào xuất hiện là gì trong một trường hợp nhất định? Tôi không biết. Tôi phải giả thuyết. Giả thuyết đó được gọi là gì? Loại giả thuyết nào mà tôi phải dùng? Tôi cần một giả thuyết-

-- 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.

--của phân bố thống kê của các đầu vào. Ngược lại, thời gian dự kiến sẽ không có ý nghĩa bởi vì tôi không biết xác suất của các thứ là bao nhiêu. Để tính xác suất, bạn cần một số giả thuyết và bạn phải phát biểu những giả thuyết này rõ ràng. **Một trong những giả thuyết tổng quát nhất là tất cả các đầu vào có khả năng bằng nhau.** Đó được gọi là phân bố đều. Mỗi đầu vào kích thước n có khả năng như nhau, tương tự như nhau. Nhưng bạn có thể giả thuyết theo những cách khác, và có thể không phải tất cả chúng đều đúng. Cái này phức tạp hơn nhiều, như bạn có thể thấy. May thay, tất cả các bạn đều có kiến thức về xác suất vững. Và vì vậy chúng ta sẽ không gặp khó khăn gì trong việc xác định các vấn đề xác suất để giải quyết kì vọng và các thứ tương tự thế.

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.

Nếu bạn không rành, đây là lúc để đi và nói, được, có lẽ tôi nên học lớp xác suất vì môn đó là tiên quyết cho môn này. Lúc trước tôi đang đề cập đến phân tích trường hợp xấu nhất. Và điều này tôi khẳng định là không có thật. Không có thật. Không tốt. Tại sao phân tích trường hợp xấu nhất không có thật? Vâng? Có lẽ trường hợp xấu nhất chưa bao giờ xảy ra. Thực sự, nó lí thú vì đối với bài toán sắp xếp, những thứ phổ biến nhất được sắp xếp là những thứ đã được sắp xếp thú vị rồi, hoặc ít nhất là gần như được sắp xếp. Chẳng hạn, một trong những thứ được sắp xếp thông thường nhất là mã số kiểm tra ở các ngân hàng. Chúng có khuynh hướng đi vào theo thứ tự giống như chúng được viết.

They're sorting things that are almost always sorted. I mean, it's good. When upper bound, 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.

Họ sẽ sắp xếp những thứ hầu như luôn luôn đã được sắp xếp. Ý tôi là điều đó tốt. Khi ràng buộc lớn hơn, không phải ràng buộc nhỏ hơn? Vâng, bạn muốn tạo ra sự bảo đảm. Và vì vậy tại sao điều này không phải là một sự bảo đảm? Bạn đang ở trên thứ gì đó, nhưng chúng ta cần một ít sự chính xác hơn ở đây. Tôi có thể lừa như thế nào? Vâng? Vâng, bạn có thể bị lừa. Bạn bị lừa. Bạn nhận bất kì một thuật toán chậm mà

bạn muốn và chỉ kiểm tra một số đầu vào nào đó, và nếu nó là đầu vào đó, thì ngay lập tức bạn nói vâng, được, đây là câu trả lời. Và do vậy đó là một trường hợp tốt nhất.

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.

Nhưng tôi không bảo bạn bất cứ điều gì về phần đông nhất những gì sẽ xảy ra. Vì vậy, bạn có thể bị lừa với giải thuật chậm làm việc nhanh trên đầu vào nào đó. Nó thực sự không làm nhiều cho bạn vì vậy thông thường chúng ta không lo lắng về điều đó. Hãy xét. Thời gian trường hợp xấu nhất chèn trực tiếp là gì? Bây giờ chúng ta nhận vào một vài loại vấn đề vui. Trước hết, nó phần nào phụ thuộc vào máy tính mà bạn đang chạy. Máy tính của ai, đúng không? Nó là một siêu máy tính hay là đồng hồ đeo tay của tôi? Chúng có khả năng tính toán khác nhau.

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.

Và khi chúng ta so sánh các giải thuật, chúng ta so sánh chúng thông thường là về tốc độ tương đối. Trong trường hợp này, bạn so sánh hai thuật toán trên cùng một máy. Bạn có thể tranh luận, vâng, máy gì không quan trọng vì tôi sẽ chỉ xét tốc độ tương đối của chúng. Nhưng, tất nhiên, tôi có lẽ cũng quan tâm đến tốc độ tuyệt đối. Có phải một thuật toán thực sự tốt hơn không phụ thuộc vào việc nó chạy trên máy nào? Và thế điều này phần nào gây bối rối bởi vì làm sao tôi có thể nói về thời gian trường hợp xấu nhất của một thuật toán của một phần mềm mà lại không đề cập đến phần cứng bởi vì, rõ ràng, nếu nó đã được chạy trên máy tốt hơn, thì tất nhiên nó sẽ chạy nhanh hơn. Vì vậy, đây là lúc mà bạn nhận được ý tưởng lớn về thuật toán.

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.

Đó là lí do tại sao thuật toán là một lĩnh vực lớn như thế, tại sao nó sinh ra những công ti như Google, như Akamai, như Amazon. Tại sao phân tích giải thuật, qua lịch sử máy tính, đã đạt được thành công lớn như thế, là khả năng của chúng ta nắm vững và có thể lấy những gì nhìn vẻ bề ngoài có vẻ là một tình huống lộn xộn, phức tạp và đơn giản hóa nó để có thể làm một số tính toán. Và ý tưởng đó được gọi là phân tích tiệm cận.



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.

Và ý tưởng cơ bản của phân tích tiệm cận là bỏ qua các hằng số phụ thuộc vào máy - - và thay vì thời gian chạy thực sự, hãy xét sự gia tăng của thời gian chạy. Vì vậy, chúng ta không xét thời gian chạy thực sự. Chúng ta xét sự tăng. Hãy xem những gì mà chúng ta muốn nói qua điều đó. Đây là một ý tưởng lớn. Chúng ta sẽ sử dụng một vài bài giảng tìm hiểu hàm ý của điều đó và về cơ bản sẽ làm nó trong suốt học kì.

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.

Và trong vai trò của một kĩ sư thực hành, bạn sẽ làm nó mọi lúc. Để làm điều đó, chúng ta chấp nhận một số ý niệm sẽ giúp chúng ta. Đặc biệt, chúng ta sẽ thông qua khái niệm tiệm cận. Đa số các bạn đã hiểu một số loại khái niệm tiệm cận. Có lẽ vài bạn không biết, nhưng về cơ bản bạn nên biết một chút. Cái mà chúng ta sẽ dùng trong lớp này chú ý là khái niệm theta.

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 than  $n^2$ . 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.

Và khái niệm theta là khái niệm khá dễ để nắm vững bởi vì tất cả những gì bạn làm là, từ công thức, chỉ bỏ qua những số hạng bậc thấp và bỏ qua các hằng số đầu. Chẳng hạn, nếu tôi có một công thức là  $3n^3 = 90n^2 - 5n + 6046$ , tôi giả sử, vâng, tôi bỏ số hạng bậc thấp nào? Vâng,  $n^3$  là một số hạng lớn hơn  $n^2$ . Tôi sẽ bỏ tất cả những số hạng này và bỏ qua hằng số đầu, vì vậy tôi nói đó là theta ( $n^3$ ). Điều đó khá dễ. Vì vậy đó là khái niệm theta. Bây giờ, đây là một thủ thuật thao tác khái niệm theta. Đó thực sự là một định nghĩa toán học của cái này, cái mà chúng ta sẽ đề cập trong lần tới, nó là định nghĩa theo tập hợp các hàm. Và, bạn sẽ có trách nhiệm, đây là một lớp về cả toán học và khoa học máy tính.

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 notation and of its related O notation and omega notation.

Qua khóa học bạn sẽ chịu trách nhiệm cho cả tính chính xác toán học như thể nó là một khóa học toán học và kiến thức căn bản về kĩ thuật bởi vì nó là một khóa học kĩ thuật. Chúng ta sẽ làm cả hai. Đây là cách tiếp cận chuyên môn để hiểu những gì bạn làm, vì vậy bạn có trách nhiệm để có thể làm những thủ thuật này. Bạn cũng phải có nhiệm vụ hiểu định nghĩa toán học của khái niệm theta và khái niệm O liên quan của nó và kí hiệu omega.

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.

Nếu tôi xem như  $n$  tiến đến vô cùng, thuật toán  $\Theta(n^2)$  luôn luôn dao động, cuối cùng, thuật toán  $\Theta(n^3)$ . Khi  $n$  càng lớn hơn, những số hạng này là gì không quan trọng nếu tôi đang mô tả hành vi chính xác tuyệt đối theo công thức. Nếu tôi có thuật toán  $\Theta(n^2)$ , nó sẽ nhanh hơn khi  $n$  đủ lớn hơn thuật toán  $\Theta(n^3)$ . Những số hạng bậc thấp này không quan trọng. Hằng số đầu là gì cũng không quan trọng. Cái này sẽ luôn luôn nhanh hơn.

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.

Cho dù bạn chạy thuật toán  $\Theta(n^2)$  trên máy tính chậm và thuật toán  $\Theta(n^3)$  trên máy tính nhanh. Điều tuyệt nhất về phương pháp tiệm cận là nó thỏa mãn tiêu chí của chúng ta về việc có thể so sánh cả vận tốc tương đối và vận tốc tuyệt đối, bởi vì chúng ta có thể làm điều này bất kể nền của máy tính. Trên những nền khác nhau, chúng ta có thể nhận được những hằng số khác nhau ở đây, các hằng số phụ thuộc máy đối với thời gian chạy thực sự, nhưng nếu chúng ta xét sự gia tăng khi kích thước của đầu vào ngày càng lớn hơn, nói chung tiệm cận sẽ không thay đổi. Chẳng hạn, để tôi phác họa điều đó dưới dạng sơ đồ. Nếu tôi có  $n$  trên trục này và  $T(n)$  trên trục này.

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  $n_0$  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  $n_0$  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.

Điều này có thể là, chẳng hạn, thuật toán Theta ( $n^3$ ) và đây có lẽ là thuật toán Theta( $n^2$ ). Luôn luôn sẽ có một điểm  $n_0$  nào đó mà ở đó mọi thứ lớn hơn thuật toán Theta( $n^2$ ) sẽ rẻ hơn thuật toán Theta( $n^3$ ) bất kể sự thuận lợi ban đầu bạn cho nó là bao nhiêu theo tốc độ của máy tính mà bạn đang chạy. Bây giờ, từ quan điểm kĩ thuật, có một số tiêu chí mà chúng ta phải giải quyết bởi vì thỉnh thoảng có thể  $n_0$  quá lớn đến nỗi các máy tính không đủ lớn để chạy bài toán. Tuy thế, đó là lí do tại sao chúng ta quan tâm đến một số thuật toán chậm hơn, vì một số thuật toán chậm hơn, cho dù chúng gần như không chậm hơn, ý tôi là gần như chúng sẽ chậm hơn.

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.

Chúng có lẽ vẫn còn nhanh hơn trên kích thước hợp lí của các thứ. Và vì vậy chúng ta phải cân bằng cả kiến thức toán học của chúng ta với kiến thức nền tảng kĩ thuật của chúng ta để làm công việc lập trình tốt hơn. Vì vậy, chỉ phân tích các thuật toán không tự động làm cho bạn trở thành nhà lập trình giỏi. Bạn cũng cần phải học cách để lập trình và dùng những công cụ này trong thực tế để hiểu khi nào chúng thích đáng và khi nào chúng không thích đáng. **Người ta nói rằng.**

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.

Nếu bạn muốn trở thành một nhà lập trình giỏi, bạn chỉ cần lập trình thường xuyên trong vòng hai năm, bạn sẽ trở thành một nhà lập trình xuất sắc. Nếu bạn muốn trở thành một nhà lập trình đẳng cấp thế giới, bạn có thể lập trình hàng ngày trong khoảng 10 năm, hoặc bạn có thể lập trình hàng ngày trong khoảng 2 năm và học qua một lớp giải thuật. Chúng ta hãy quay trở lại những gì chúng ta đang làm, đó là phân tích sự chèn trực tiếp. Chúng ta sẽ xét trường hợp xấu nhất. Nó, như chúng ta đã đề cập từ trước, là khi đầu vào bị sắp xếp ngược. Những số lớn nhất nằm ở phía trước và những số nhỏ nằm ở phía sau vì bây giờ mỗi lần bạn sắp xếp bạn phải lê bước trên từng cái.

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 beauty of the method is that it causes all these things that are real distinctions to sort of vanish.

Bạn có thể viết ra thời gian chạy bằng cách xét sự lồng nhau của các vòng lặp. Những gì chúng ta làm là chúng ta cộng lại. Những gì chúng ta giả sử là mỗi hoạt động, mỗi hoạt động cơ bản sẽ mất một lượng thời gian không đổi. Nhưng chúng ta không cần phải lo lắng về việc hằng số đó bằng bao nhiêu vì chúng ta sẽ làm phân tích tiệm cận. Như tôi nói, sự đẹp đẽ của phương pháp là nó làm cho tất cả những thứ này khác biệt thực sự với sự sắp xếp triệt tiêu.

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  $j$ ?

Chúng ta phần nào xét chúng từ 30 nghìn bộ còn hơn từ 3 milimet hoặc thứ gì đó. Mỗi hoạt động này sẽ phần nào là các hoạt động cơ bản. Một cách để nghĩ về điều này, theo sự đếm các hoạt động, đếm tham chiếu bộ nhớ. Bạn thực sự truy cập biến nào đó bao nhiêu lần? Đó là cách nghĩ khác về mô hình này. Khi chúng ta làm điều đó, vâng, chúng ta sẽ xét kĩ vòng lặp này,  $j$  chạy từ 2 đến  $n$ , và sau đó chúng ta sẽ cộng thêm công việc mà chúng ta làm trong vòng lặp. Chúng ta có thể phần nào viết ra điều đó trong toán học như tổng của  $j$  bằng 2 đến  $n$ . Và sau đó công việc sẽ tiếp tục trong vòng lặp này là gì? Vâng, công việc tiếp tục trong vòng lặp này biến đổi, nhưng trong trường hợp xấu nhất có bao nhiêu hoạt động sẽ diễn ra ở đây cho mỗi giá trị của  $j$ ?

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?

Đối với giá trị cho trước của  $j$ , có bao nhiêu công việc diễn ra trong vòng lặp này? Ai đó nói cho tôi biết xem nào? Tiệm cận. Nó bằng  $j$  nhân một hằng số nào đó, vì vậy nó bằng  $\theta(j)$ . Vì vậy có  $\theta(j)$  công việc diễn ra ở đây bởi vì vòng lặp này khởi đầu với  $i$  bằng  $j$  trừ 1, và do đó nó sẽ làm đúng một lượng công việc cho mỗi bước của giá trị của  $i$ , và  $i$  đang chạy từ  $j$  trừ 1 xuống 0. Vì vậy, chúng ta có thể nói đó là  $\theta(j)$  công việc đang diễn ra. Các bạn theo kịp không?

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.

Vâng. Và chúng ta có công thức mà chúng ta có thể ước lượng. Sự ước lượng là gì? Nếu tôi muốn đơn giản hóa công thức này, nó sẽ bằng cái gì? Xin lỗi. Nhắc lại nào. Vâng. Được. Đó chỉ là  $\Theta(n^2)$ , tốt. Vì khi bạn nói tổng của các số liên tiếp, bạn muốn nói gì? Số hạng toán học mà chúng ta có cho cái đó là gì để chúng ta có thể truyền? Bạn phải biết những thứ này để bạn có thể truyền. Nó được gọi là dãy gì? Nó thực sự là một chuỗi, nhưng điều đó đúng. Loại chuỗi này được gọi là gì? Chuỗi số học, tốt. Wow, chúng ta có một số người nhạy bén có thể truyền.

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.

Đây là chuỗi số học. Về cơ bản bạn cộng  $1 + 2 + 3 + 4$ , một vài hằng số ở đó, nhưng về cơ bản nó là  $1 + 2 + 3 + 4 + 5 + 6$  lên đến  $n$ . Đó là  $\Theta(n^2)$ . Nếu bạn không biết toán học này, có một chương trong sách, hoặc có thể bạn đã học rồi. Chuỗi Erythematic. Mọi người có sự hồi tưởng mơ hồ này. Oh, vâng. Tốt. Bạn phải học những thủ thuật này. Chúng ta sẽ nói về chúng một ít trong lần tới, nhưng bạn phải học những thủ thuật theta này cho những gì làm việc với theta. Và bạn phải rất cẩn thận vì theta là một kí hiệu yếu. Một kí hiệu mạnh là cái giống như các kí hiệu Leibniz từ phép tính mà ở đó quy tắc dây chuyền chỉ hủy bỏ hai thứ.

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.

Thật hoang tưởng nếu bạn nghĩ bạn có thể triệt tiêu theo quy tắc dây chuyền. Và kí hiệu Leibniz chỉ biểu diễn điều đó để cho bạn có thể thao tác. Kí hiệu Theta không giống như thế. Nếu bạn nghĩ nó là như thế bạn sẽ gặp khó khăn. Bạn cần phải nghĩ những gì đang diễn ra theo kí hiệu Theta. Và nó là một kí hiệu mô tả hơn là là một kí hiệu thao tác. Có những thủ thuật mà bạn có thể thực hiện với nó, nhưng nếu bạn không hiểu những gì đang thực sự diễn ra trong kí hiệu theta bạn sẽ tự đưa mình vào khó khăn.

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  $n$ .

Và lần tới chúng ta sẽ nói thêm một ít về kí hiệu theta. Chèn trực tiếp nhanh phải không? Vâng, hóa ra là đối với  $n$  nhỏ nó nhanh vừa phải. Nhưng nếu  $n$  lớn thì nó không nhanh gì cả. Vì vậy, tôi sẽ cho bạn một thuật toán nhanh hơn. Nó được gọi là sắp xếp kiểu trộn. Tôi tự hỏi có nên bỏ phương pháp chèn trực tiếp hay không. Tại sao không. Tôi sẽ viết về điều này sau, vì vậy nếu bạn đang chép, hãy để lại một khoảng trống bên trái. Đây là sắp xếp kiểu trộn của một mảng  $A$  từ 1 đến  $n$ .

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.

Và về cơ bản nó gồm ba bước. Nếu  $n$  bằng 1 chúng ta hoàn thành. Sắp xếp một yếu tố, nó đã được sắp xếp rồi. Được rồi. Thuật toán đệ quy. Ngược lại, những gì chúng ta làm là chúng ta sắp xếp đệ quy A từ 1 đến ceiling của  $n$  trên 2. Và mảng A của ceiling  $n$  trên 2 cộng 1 đến  $n$ . Vì vậy, chúng ta sắp xếp hai nửa của đầu vào. Và sau đó, ba, chúng ta lấy hai list đã làm xong và trộn chúng lại.

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  $A[\lfloor n/2 \rfloor]$  and  $A[\lfloor n/2 \rfloor + 1]$  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.

Và để làm điều đó, chúng ta dùng thủ tục con trộn mà tôi sẽ chỉ cho bạn. Thủ tục con then chốt ở đây là trộn, và nó làm việc như thế này. Tôi có hai list. Giả sử một trong số chúng là 20. Tôi sẽ làm điều này theo thứ tự ngược. Tôi đã sắp xếp cái này giống như thế này. Và sau đó tôi sắp xếp một cái khác. Nhưng dù sao đi nữa, tôi không biết tại sao tôi làm nó theo thứ tự này. Đây là list khác của tôi. Tôi có hai list mà tôi đã sắp xếp. Vì vậy đây là  $A[1]$  đến  $A[\lfloor n/2 \rfloor]$  và  $A[\lfloor n/2 \rfloor + 1]$  đến  $A[n]$  theo cách mà nó sẽ được gọi trong chương trình này. Và bây giờ để trộn hai cái này, những gì tôi cần làm là tạo ra một list được sắp xếp ngoài cả hai chúng.

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.

Những gì tôi làm là đầu tiên quan sát yếu tố nhỏ nhất trong bất kì hai list đã được phân loại rồi nằm ở đâu? Nó ở một trong hai nơi này, ở đầu list thứ nhất hoặc đầu list thứ hai. Tôi nhìn hai yếu tố đó và nói cái nào nhỏ hơn? Cái này nhỏ hơn. Sau đó những gì tôi làm là xuất vào trong mảng đầu ra của tôi cái nhỏ hơn trong hai cái. Và tôi xóa nó. Và bây giờ số nhỏ nhất tiếp theo ở đâu? Và câu trả lời là nó sẽ ở đầu của một trong hai list này. Sau đó tôi xóa thằng này và đặt nó ở đây và khoanh tròn cái này. Bây giờ tôi xét hai thằng này. Cái này nhỏ hơn vì vậy tôi xuất cái đó và khoanh tròn cái đó. Bây giờ tôi xét hai thằng này, xuất 9. Vì vậy mỗi bước ở đây là một số lượng các hoạt động cố định không phụ thuộc vào kích thước của mảng tại mỗi bước.

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.

Mỗi bước riêng biệt chỉ là tôi nhìn hai yếu tố và chọn ra cái nhỏ nhất và đẩy một số con trỏ vào trong mảng để cho tôi biết được đầu hiện tại của list đó đang ở đâu. Và vì vậy, do đó, thời gian là bậc  $n$  trên  $n$  yếu tố. Thời gian để hoàn thành cái này và trộn hai list là bậc  $n$ . Đôi khi chúng ta gọi cái này là thời gian tuyến tính vì nó không phải là bậc hai hoặc bất cứ thứ gì khác. Nó tỉ lệ với  $n$ , tỉ lệ với kích thước đầu vào. Nó là thời gian tuyến tính. Nó lướt qua và chỉ làm hoạt động đơn giản này, chỉ tiến dần trên những list này, và cuối cùng về cơ bản tôi đã hoàn thành  $n$  hoạt động, hoạt động thứ  $n$  mỗi cái trong chúng mất một khoảng thời gian không đổi.

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.

Đó là tổng của thời gian bậc  $n$ . Mọi người có đồng ý không? Vâng. Vì vậy, đây là một chương trình đệ quy. Bây giờ chúng ta thực sự có thể viết ra những gì được gọi là đệ quy cho chương trình này. Cách mà chúng ta làm điều đó là giả sử thời gian để sắp xếp  $n$  yếu tố là  $T(n)$ . Vậy mất bao lâu để làm bước 1? Đó chỉ là một thời gian không đổi. Chúng ta chỉ kiểm tra xem  $n$  có bằng 1 hay không, và nếu nó bằng 1 chúng ta quay lại. Điều đó không phụ thuộc vào kích thước của bất cứ thứ gì mà chúng ta đang làm. Nó chỉ lấy một số chương trình máy nào đó trên máy nào đó và chúng ta nói nó là thời gian không đổi. Chúng ta gọi cái đó là theta một. **Đây thực sự là một ít lạm dụng nếu bạn dính vào nó.**

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.

Và lí do là bởi vì thông thường để nói nó bạn cần nói những gì nó đang tăng cùng. Tuy thế, chúng ta dùng cái này như một sự lạm dụng của kí hiệu chỉ để nói nó là hằng số. Vì



vậy đó là sự lạm dụng chỉ để người ta biết. Nhưng nó đơn giản hóa sự việc nếu tôi chỉ viết  $\theta$  một. Và về cơ bản nó nói về cùng một thứ. Bây giờ chúng ta sắp xếp đệ quy hai cái này. Tôi có thể mô tả điều đó như thế nào? Thời gian để làm điều này, tôi có thể mô tả đệ quy là  $T$  của ceiling  $n$  trên 2 cộng  $T$  của  $n$  trừ ceiling của  $n$  trên 2. Đó thực sự là phần nào sự trộn, vì vậy những gì chúng ta sẽ làm là tùy tiện và viết  $2T(n/2)$ . Điều này chỉ là sự tùy tiện của chúng ta.

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.

Và vào thứ 6 chúng ta sẽ thấy trong phần trả lời miệng rằng nó là đúng để tùy tiện. Đó là thứ tuyệt vời về giải thuật. Miễn là bạn nghiêm khắc và chính xác, bạn có thể tùy tiện như bạn muốn. Điều này tùy tiện bởi vì tôi không lo lắng về những gì xảy ra, vì hóa ra là nó không tạo ra sự khác biệt. Và chúng ta sẽ thực sự thấy điều đó đúng. Và, cuối cùng, tôi phải trộn hai list đã được sắp xếp có tổng cộng  $n$  yếu tố. Và chúng ta chỉ phân tích cái đó dùng thủ tục con trộn. Và điều đó làm chúng ta mất thời gian  $\theta n$ .

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  $\theta n$  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.

Bây giờ điều đó cho phép chúng ta viết hồi quy để thực hiện sắp xếp trộn. Nó nói rằng  $T_n$  bằng  $\theta 1$  nếu  $n$  bằng 1 và  $2T$   $n$  trên 2 cộng  $\theta n$  nếu  $n$  lớn hơn 1. Bởi vì hoặc là tôi sẽ làm bước 1 hoặc là tôi sẽ làm tất cả các bước 1, 2 và 3. Ở đây tôi sẽ làm bước 1 và tôi quay lại và tôi hoàn thành. Hoặc nếu không tôi sẽ làm bước 1, tôi không trở lại, và sau đó tôi cũng làm bước hai và ba. Vì vậy, tôi cộng những cái này với nhau. Tôi có thể nói  $\theta n$  cộng  $\theta 1$ , nhưng  $\theta n$  cộng  $\theta 1$  cũng chỉ là  $\theta n$  vì  $\theta 1$  là số hạng bậc nhỏ hơn  $\theta n$  và tôi có thể bỏ qua nó. Nó hoặc bằng  $\theta 1$  hoặc bằng  $2T$   $n$  trên 2 cộng  $\theta n$ . Bây giờ, thông thường chúng ta sẽ không viết cái này.

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.

Thông thường chúng ta bỏ qua nó. Nếu nó không tạo ra sự khác biệt cho nghiệm hồi quy, chúng ta sẽ thường bỏ qua những trường hợp nền không đổi. Trong giải thuật, nói chung nó không giống chính xác như trong toán học, nhưng trong giải thuật nếu bạn chạy thứ gì đó trên đầu vào kích thước không đổi nó luôn luôn mất một khoảng thời gian không đổi. Vì vậy chúng ta không phải lo lắng về việc giá trị này bằng bao nhiêu. Và hóa ra là nó thực sự không ảnh hưởng đến nghiệm đệ quy tiệm cận.



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.

Cách chúng ta giải một hồi quy giống như thế này như thế nào? Bây giờ tôi có  $T_n$  được biểu diễn theo  $T$   $n$  trên 2. Điều đó ở trong sách và nó cũng ở trong chương 2. Chúng ta sẽ học chương 2 để giải cái đó, nhưng vào lúc đó những gì tôi sẽ làm là cho bạn một phương pháp trực quan để hiểu cái này quan trọng như thế nào, nó là một trong những kĩ thuật mà chúng ta sẽ nói chi tiết trong lần tới. Nó được gọi là kĩ thuật cây đệ quy. Và tôi sẽ dùng nó cho đệ quy thực gần như giống  $2T(n/2)$ , nhưng tôi sẽ diễn đạt thật sự rõ ràng, bởi vì tôi muốn bạn thấy nó xuất hiện ở đâu, cộng hằng số nào đó nhân  $n$  ở đây  $c$  là hằng số lớn hơn 0. Vì vậy, chúng ta sẽ xét đệ quy này với trường hợp cơ bản bậc 1.

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.

Tôi sẽ chỉ tạo ra một hằng số ở đây, ràng buộc lớn hơn trên hằng số là rõ ràng hơn là ngấm ngấm. Và cách bạn làm cây đệ quy là như sau. Bạn bắt đầu bằng cách viết ra ở vế trái của hồi quy. Và sau đó những gì bạn làm là bạn nói, à, cái đó tương đương, và bây giờ hãy viết nó như một cây. Tôi làm  $cn$  công việc cộng bây giờ tôi sẽ phải làm công việc trên mỗi hai đứa con của tôi.

$T$  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.

$T$   $n$  trên 2 và  $T$   $n$  trên 2. Nếu tôi cộng những gì ở đây, tôi nhận được cái này vì đó là những gì hồi quy nói,  $T(n)=2T(n/2)+cn$ . Tôi có  $2T(n/2)+cn$ . Sau đó tôi làm lại điều đó một lần nữa. Tôi có  $cn$  ở đây. Bây giờ tôi có  $cn/2$  ở đây. Và bây giờ mỗi cái này có  $T(n/4)$ . Và mỗi cái này có  $T(n/4)$ . Và cái này có  $T(n/4)$ . Và tôi tiếp tục làm điều đó, chấm nguy hiểm, chấm, chấm. Và, nếu tôi tiếp tục làm điều đó, tôi sẽ kết thúc với kết quả giống như thế này.

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?

Và tôi tiếp tục đi xuống cho đến khi tôi đến một lá. Và một lá, về cơ bản tôi có  $T(1)$ . Đó là  $T(1)$ . Và vì vậy câu hỏi đầu tiên tôi đặt ra ở đây là, chiều cao của cây này bằng bao nhiêu? Vâng. Nó bằng  $\log n$ . Nó thực sự rất gần với  $\log n$  bởi vì tôi khởi đầu tại ngọn với  $n$  và sau đó tôi đi đến  $n/2$  và  $n/4$  và tất cả đường đi xuống cho đến khi tôi gặp 1. Số phân nửa của  $n$  cho đến khi tôi gặp 1 là  $\log n$  vì vậy ở đây độ cao là  $\log n$ . Nó đúng nếu nó là hằng số nhân  $\log n$ . Không sao cả. Tiện thể, có bao nhiêu lá trong cây này?

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.

Có bao nhiêu lá mà cây này có? Vâng. Số lá, một lần nữa, thực sự khá gần, Nó thực sự bằng  $n$ . Nếu bạn đem nó tất cả đường đi xuống. Chúng ta hãy làm một số giả thuyết đơn giản.  $n$  là lũy thừa hoàn toàn của 2, vì vậy nó là lũy thừa nguyên của 2. Do đó cái này đúng bằng  $\log n$  để đi xuống  $T(1)$ . Và sau đó có đúng  $n$  lá, vì số lá ở đây, số nhánh ở mức này là 1, 2, 4, 8.

And if I go down height  $h$ , I have  $2^h$  leaves,  $2^{\log n}$ , 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?  $\Theta(n)$ .

Và nếu tôi đi xuống độ cao  $h$ , tôi có  $2^h$  lá,  $2^{\log n}$ , nó đúng bằng  $n$ . Chúng ta làm toán ở đây, đúng không? Bây giờ chúng ta hãy tìm hiểu xem có bao nhiêu công việc, nếu tôi xét cộng mọi thứ trong cây này tôi sẽ được  $T(n)$ , vì vậy chúng ta hãy cộng cái đó. Vâng, chúng ta hãy cộng theo từng mức. Chúng ta có bao nhiêu cái ở mức đầu tiên? Đúng bằng  $cn$ . Nếu tôi cộng mức 2, tôi có bao nhiêu?  $cn$ . Còn nếu tôi cộng mức thứ ba thì sao?  $cn$ . Còn nếu tôi cộng tất cả các lá?  $\Theta(n)$ .

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.

Không nhất thiết là  $cn$  vì trường hợp biên có một hằng số khác. Nó thực sự bằng  $\Theta(n)$ , nhưng  $cn$  mọi đường ở đây. Nếu tôi cộng lượng tổng, nó bằng  $cn$  nhân  $\log n$ , bởi vì đó là chiều cao, đó là bao nhiêu  $cn$  mà tôi có ở đây, cộng  $\Theta(n)$ . Và cái này là số hạng bậc lớn hơn cái này, vì vậy có thể cái này, bỏ hằng số, nó bằng  $\Theta(n \lg n)$ . Và  $\Theta(n \lg n)$  tiệm cận nhanh hơn  $\Theta(n^2)$ . Vì vậy, sắp xếp kiểu trộn, trên một kích thước đầu vào đủ lớn, sẽ thành chèn trực tiếp.

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.

Sắp xếp kiểu trộn là một thuật toán nhanh hơn. Xin lỗi, các bạn, tôi không nhận ra bạn không thấy chỗ đó. Bạn phải nói thẳng nếu bạn không thấy. Vì vậy đây là thuật toán nhanh hơn bởi vì  $\theta(n \lg n)$  tăng chậm hơn  $\theta(n^2)$ . Và sắp xếp kiểu trộn tiến tiệm cận với chèn trực tiếp. Cho dù bạn chạy chèn trực tiếp trên siêu máy tính, ai đó chạy sắp xếp kiểu trộn trên máy tính cá nhân với đầu vào đủ lớn sẽ đánh bại hoàn toàn chúng bởi vì thực sự  $n^2$  luôn lớn hơn  $n \log n$  một khi bạn nhận được những  $n$  lớn. Và, trong thực tế, sắp xếp kiểu trộn có khuynh hướng thắng ở đây đối với  $n$  lớn hơn, giả sử, 30 hoặc v.v...

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.

Nếu bạn có đầu vào rất nhỏ ví dụ cỡ 30 yếu tố, chèn trực tiếp là loại thuật toán đúng đắn để dùng. Nhưng sắp xếp kiểu trộn sẽ nhanh hơn nhiều thậm chí đối với những trường hợp mà chỉ có vài tá yếu tố. Nó thực sự là một thuật toán nhanh hơn. Đó là cách sắp xếp bài học, đúng không? Hãy nhớ nhận bài tập kiểm tra miệng của bạn và tập trung vào bài kiểm tra miệng vào thứ sáu. Vì chúng ta sẽ nghiên cứu chi tiết nhiều thứ mà tôi để lại trên bảng ở đây. Và hẹn gặp lại các bạn vào thứ hai tuần tới.