**About the course**

In this video I'll talk about various aspects of the course, the topics that we'll cover, the kinds of skills you can expect to acquire, the kind of background that I expect, the supporting materials and the available tools for self assessment.

在这个视频中，我将谈论这个课程的方方面面，我们主要介绍的话题，你可以期待获得各种技能，我所期待的那种背景，支持材料和可用于自我评估的工具

Let's start with the specific topics that this course is going to cover.

让我们从本课程将涵盖的特定主题开始

The course material corresponds to the first half of the ten week Stanford course.

课程材料对应斯坦福大学十周课程的前半部分

It's taken by all computer science undergraduates, as well as many of our graduate students.

本课程供所有计算机科学的本科学生，以及很多我们的研究生学习

There will be five high level topics, and at times these will overlap.

课程有五个高级主题，有时这些主题会有重叠

The five topics are first of all, the vocabulary for reasoning about algorithm performance, the design and conquer algorithm design paradigm, randomization and algorithm design, primitives for reasoning about graphs, and the use and implementation of basic data structures.

这五个主题首先是，算法性能推理的词汇，分而治之算法设计范式，随机化和算法设计，图形推理原语，以及基本数据结构的使用和实现

The goal is to {provide an introduction to and basic literacy} in each of these topics.

目标是在每个主题中提供介绍和基本读写能力

Much, much more could be said about each of them, than we'll have time for here.

关于它们每一个主题，我们可以说的比我们在这里的时间多得多

The first topic is the shortest, and probably also the driest.

第一个主题是最短的，也有可能是最枯燥的

But it's a prerequisite for thinking seriously about the design and analysis of algorithms.

但是，它是认证思考算法设计和分析的先决条件

The key concept here is big-O notation, which, conceptually, is a modeling choice about the granularity with which we measure a performance metric like the running time of an algorithm.

这里的关键概念是“大O表示法“，从概念上讲，它是关于粒度的建模选择，我们用它来测量性能的指标，如算法的运行时间

It turns out that the sweet spot for clear high level thinking about algorithm design, is to ignore constant factors and lower-order terms.

结果证明，关于算法设计的清晰高层次思考的最佳点是忽略常数因子和低阶项

And to concentrate on how well algorithm performance scales with large input sizes.

并关注算法在大输入量下的性能如何伸缩

Big O notation is the way to mathematize this sweet spot.

大O表示法是量化表示这个最佳点的方法

Now, there's no one silver bullet in algorithm design.

现在，算法设计中没有一颗“银色子弹”

No single problem solving method that's guaranteed to unlock all of the

computational problems that you're likely to face.

没有单一的问题解决方法能够保证解决你可能遇到的所有计算问题

That said, there are a few general algorithm design techniques.

这也就是说，有一些通用的算法设计技术

High level approaches to algorithm design that find successful application across a range of different domains.

高级算法设计方法，可以在一系列不同的域中找到成功的应用

These relatively widely applicable techniques are the backbone of a general algorithms course like this one.

这些相对广泛适用的技术是向本课程这样的通用算法课程的核心内容

In this course, we'll only have time to deeply explore one such algorithm design paradigm, namely that of the divide and conquer algorithms.

在本课程中，我们只有时间深入探索一种这样的算法设计范例，即分而治之的算法

In the sequel course as we'll discuss, there's two other major algorithms on paradigms to get covered.

在我们即将讨论的续集课程中，还有另外两种关于范例的主要算法会被介绍

But for now, divide and conquer algorithm, the idea is to first break the problem into smaller problems which then gets solved recursively, and then to somehow quickly combine the solutions to the sub problems into one for the original problem that you actually care about.

但就目前而言，分而治之的算法，其思路是，首先讲问题分解成小问题，然后递归求解，接着以某种方式快速的把子问题的解决方法合并，从而解决实际上你关心的那个原始问题

So for example, in the last video.

所以，举个例子，在上一个视频中

We saw two algorithms of this sort, two divide and conquer algorithms from multiplying two large integers.

我们看到了两个这个种类的算法，两个分而治之算法来将两个大整数相乘

In later videos we will see a number of different applications.

在以后的视频中，我们将看到若干不同的应用

We'll see how to design fast divide and conquer algorithms for problems ranging from sorting to matrix multiplication to nearest neighbor-type problems and computation of geometry.

我们将看到如何设计快速的分而治之算法的问题，从排序到矩阵乘法，再到最近邻类型问题和几何计算

In addition, we'll cover some powerful methods for reasoning about the running time of recursive algorithms like these.

另外，我们将介绍一些强大的方法来推理出这些递归算法的运行时间

As for the third topic.

至于第三个话题

A randomized algorithm is one that, in some sense, flips coins while it executes. That is, a randomized algorithm will actually have different executions if you run it over and over again on a fixed input.

随机算法在执行时，在某种意义上说，是掷硬币，也就是说，如果在固定输入下反复运行，随机算法实际上会有不同的执行。

It turns out, and this is definitely not intuitive, that allowing randomization internal to an algorithm, often leads to simple, elegant, and practical solution to various computational problems.

事实证明，这绝对不是直观的，允许算法内部的随机化，通常能让各种计算问题通向简单、优雅和实用的解决方法

The canonical example is randomized quick sort, and that algorithm and analysis we will cover in detail in a few lectures.

经典的例子是快速随机排序，我们将在一些讲座中介绍这个算法和分析

{Randomized primality testing} is another killer application that we'll touch on. And we'll also discuss a randomized approach to graph partitioning. And finally we'll discuss how randomization is used to reason about hash functions and hash maps.

随机素性测试是另外一个我们将要提及的杀手级应用。我们还将讨论图形分区的随机方法。最后，我们将讨论如何用随机化推理散列函数和散列映射的

One of the themes of this course, and one of the concrete skills that I hope you take away from the course, is, literacy with a number of computational primitives for operating on data, that are so fast, that they're, in some sense, essentially

free.

本课程的主题之一，也是我希望你从课程中学到的具体技巧之一，是用于操作数据的大量的计算原语的读写能力，它们是如此之快，以至于他们在某种意义上来说，本质上是自由的

That is, the amount of time it take to invoke one of these computational primitives is barely more than the amount of time ~~you're~~ already spending just examining or reading the input.

也就是说，调用其中一个计算原语所花费的时间几乎不会超过它刚才检查或读取输入所花费的时间

When you have a primitive which is so fast, that the running time is barely more than what it takes to read the input, you should be ready to apply it.

当你拥有一个如此之快的原语时，运行时间几乎不会超过它读取输入的时间，你应该准备应用它

For example, in a preprocessing step, whenever it seems like it might be helpful. It should just be there on the shelf // waiting to be applied at will.

举个例子，在预处理阶段，每当它看起来可能有帮助，它就应该被放在架子上，等待被随时应用

Sorting is one canonical example of a very fast, almost for-free primitive of this form. But there are ones that operate on more complex data as well.

排序是这种形式的一种非常快速，几乎免费的原语的一个经典例子。但是也有一些对更复杂数据的操作

So recall that a graph is a data structure that has, on the one hand, vertices, and

on the other hand, edges. Which connects pair of vertices. Graphs model, among any other things, different types of networks.

所以请记住，图形也是一种数据结构，一方面是顶点，一方面是边缘，它们连接一对顶点，图表模型，以及其他任何东西，不同类型的网络

So even though graphs are much more complicated than mere arrays, there's still a number of blazingly fast primitives for reasoning about their structure.

因此，尽管图形比单纯的数组复杂的多，但仍然有一些用来推理它们的结构的非常快的原语

In this class we'll focus on primitives for competing connectivity information and also shortest paths.

在本课程中，我们将聚焦竞争连接信息的原语和最短路径

We'll also touch on how some primitives have been used to investigate the structure of information in social networks.

我们还将讨论如何使用一些原语来研究社交网络中的信息结构

Finally, data structures are often a crucial ingredient in the design of fast algorithms.

最后，数据结构通常也是快速算法设计的重要组成部分

A data structure's responsible for organizing data in a way that supports fast queries.

数据结构负责以支持快速查询的方式来组织数据

Different data structures support different types of queries.

不同的数据结构支持不同类型的查询

I'll assume that you're familiar with the structures that you typically encounter in a basic programming class including arrays and vectors. Lists, stacks, and queues.

我假定你熟悉在基础编程课中通常遇到的结构，包括数组和向量、列表、堆栈以及队列

Hopefully, you've seen at some point both trees and heaps, or you're willing to read a bit about them outside of the course, but we'll also include a brief review of each of those data structures as we go along.

希望你在某些时候已经看过树和堆，或者你愿意在课程外阅读一些关于它们的东西，但是，我们仍然会在我们进行时简要回顾每个数据结构

There's two extremely useful data structures that we'll discuss in detail.

我们会详细讨论两个特别有用的数据结构

The first is balanced binary search trees.

第一个是平衡二叉搜索树

These data structures dynamically maintain an ordering on a set of elements, while supporting {a large number of} queries that run in {time logarithmic } in the size of the set.

这些数据结构动态的维护一组元素的排序，同时支持大量以时间对数运行的集合大小的查询

The second data structure we'll talk a fair bit about is hash tables or hash maps, which keep track of a dynamic set, while supporting extremely fast insert and lookup queries.

我们将讨论一段时间的数据结构是散列表或者散列映射，它们追踪动态集，同时支持极快速的插入和查找查询

We'll talk about some canonical uses of such data structures, as well as what's going on under the hood in a typical implementation of such a data structure.

我们将讨论这些数据结构的经典使用，以及在这个数据结构的典型实现中，底层发生的事情

There's a number of important concepts in the design and analysis of algorithms that we won't have time to cover in this five week course. Some of these will be covered in the sequel course, Design and Analysis of Algorithms II, which corresponds to the second half of Stanford's ten week course on this topic.

在算法设计和分析中，有很多重要的概念，我们没有时间在这个五周的课程中全部介绍。这其中的部分将会在后续的《算法设计与分析2》课程中介绍，该课程对应斯坦福大学关于该主题的十周课程中的后半部分。

The first part of this sequel course focuses on two more algorithm design paradigms.

该后续课程的第一部分将聚焦超过两个的算法设计范例

First of all, the design analysis of greedy algorithms with applications to minimum spanning trees, scheduling, and information theoretic coding.

首先，贪婪算法的设计分析应用于最小生成树、时序安排和信息理论编码

And secondly, the design analysis of dynamic programming algorithms with example applications being in genome sequence alignment and the shortest path protocols in communication networks.

其次，动态规划算法的设计分析，实例应用是基因组序列比对和通信网络的最短路径协议

The second part of the sequel course concerns NP complete problems, and what to do about them.

后续课程的第二部分将涉及NP完全问题，以及如何处理它们

Now, NP complete problems are problems that, assuming a famous mathematical conjecture you might have heard of, which is called the "P not equal to NP" conjecture, are problems that cannot be solved under this conjecture by any computationally efficient algorithm.

现在，NP完全问题是你可能听过的一个著名的猜想，即“P不等于NP”猜想，这个问题在这个猜想下无法通过任何从计算方面讲有效率的算法去解决

We'll discuss the theory of NP completeness, and, with a focus on what it means for you as an algorithm designer.

我们将讨论NP完全理论，以及对你这个算法设计者来说它意味着什么

We'll also talk about several ways to approach NP complete problems, including: fast algorithms that correctly solve special cases; fast heuristics with provable performance guarantees; and exponential time algorithms that are qualitatively faster than brute force search.

我们还会讨论解决NP完全问题的一些方法，包括：正确解决特定情况的快速算法；具有可证明的性能保证的快速启发式；以及定性上比蛮力搜索更快的指数时间算法

Of course there are plenty of important topics that can't be fit into either of these two five-week courses.

当然，还有大量重要的话题不适合这两个为期五周的课程

Depending on the demand, there might well be further courses on more advanced topics.

根据需求，可能会有更多高级主题的课程

Following this course is going to involve a fair amount of time and effort on your part.

完成本课程，你将需要花费大量的时间和精力

So it's only reasonable to ask: What can you hope to get out of it? What skills will you learn? Well.

所以我们有理由问：这门课程中有什么是你能期望带走的？你将学会什么技能？好。

Primarily, you know, even though this isn't a programming class per se, it should make you a better programmer.

首先，你要知道，尽管这不是一门编程课，但它可以使你成为一个更好的程序员。

You'll get lots of practice describing and reasoning about algorithms, you'll learn algorithm design paradigms, so really high level problem-solving strategies that are relevant for many different problems across different domains, and tools for predicting the performance of such algorithms.

你将得到很多关于描述和推理算法的联系，你将学到算法设计范例，所以，真正的高级问题解决策略对很多不同领域的不同问题以及预测这些算法性能的工具是有很大作用的。

You'll learn several extremely fast subroutines for processing data and several useful data structures for organizing data that can be deployed directly in your own programs.

你将学到一些用来处理数据的非常快速的子程序和一些组织数据非常有用的数据结构，这些东西你可以理解部署到你自己的程序中

Second, while this is not a math class per se, we'll wind up doing a fair amount of mathematical analysis.

第二，虽然这不是一个数学课程，但我们最终会做大量的数学分析

And this in turn will sharpen your mathematical analytical skills.

而这反过来也会磨砺你的数学分析技能

You might ask, why is mathematics relevant for a class in the design and analysis of algorithms, seemingly more of a programming class.

你也许会问，为什么数学会和一节算法设计和分析课程有关，这看来似乎更像一节编程课

Well let me be clear. I am totally uninterested in merely telling you facts or regurgitating code that you can already find on the web or in any number of good programming books.

好，来让我说清楚，我完全没有兴趣只是告诉你事实或者反刍代码，这些事实和代码你已经可以在网上或者任何好的编程书中找到

My goal here in this class, and the way I think I can best supplement the resources that you probably already have access to is to explain why things are the way they are.

我在这门课程中的目标，和我认为我可以最好的补充你可能已经获得的资源的方法，是解释为什么那些事情是那样的。

Why we analyze the algorithms in the way that we do, why various super fast algorithms are in fact super fast, and so on.

为什么我们分析算法的方法是我们所做的那些方法，为什么各种超级快的算法实际上超级快，等等

And it turns out that good algorithmic ideas usually require nontrivial mathematical analysis to understand properly.

事实证明，好的算法思想通常需要非凡的数学能力才能正确的理解

You'll acquire fundamental insights into the specific algorithms and data structures that we discuss in the course.

你将从我们在课上讨论的那些算法和数据结构中，获得基本的洞察力

And hopefully, many of these insights will prove useful, more generally, in your other work.

并且希望能够证明这些洞察力中的大部分在你其他的工作中更加有用

Third, and perhaps the most relevant for those of you who work in some other discipline: this course should help you learn how to think algorithmically.

第三，或许于那些在其他学科中工作的人最相关：这门课程将会帮助你学习如何通过算法思考

Indeed after studying algorithms it's hard enough not to see them pretty much everywhere, whether you are riding an elevator, watching a flock of birds, buying and selling stocks out of your portfolio, even watching an infant learn.

事实上，在学习了算法后，很难不发现它们真的无处不在，无论你正在乘电梯、看一群鸟、买卖投资组合中的股票，甚至是在看一个婴儿学习

As I said in the previous video algorithm thinking is becoming increasingly useful and prevalent if you are outside of computer science and technology like in biology, statistics and economics.

就像我在先前的视频中所说，算法思想正在变得日益有用和流程，即使你在计算机科技以外的，比如生物学、统计学和经济学

Fourth, if you're interested in feeling like a card carrying computer scientist, in some sense, then you'll definitely want basic literacy in all of the topics that we'll be covering.

第四，如果你有兴趣感觉自己像一个拿着卡片的计算机科学家，在某种意义上，那么你肯定会想要我们将要讨论的所有主题的基础知识。

Indeed, one of the things that makes studying algorithms so fun, is, it really feels like you're studying a lot of the greatest hits from the last 50 years of computer science.

事实上，让学习算法变得如此有趣的事情之一就是，你真的感觉自己正在研究过去50年来计算机科学领域的许多热门话题。

So, after this class, no longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra's Algorithm. Now you'll know exactly what they mean.

所以，在这节课之后，当有人在计算机科学的鸡尾酒派对上开玩笑地谈论dijkstra算法时，你将不再会感觉被排除在外。现在，你将确切的知道他们的意思

Finally, there's no question that studying this material is helpful for technical interview questions. To be clear, my sole goal here is to teach you algorithms, not to prepare you for interviews, per se.

最后，毫无疑问，学习这门课程会对技术面试问题有帮助。但需要清楚，我在这个的唯一目标是教你们算法，而不是为你准备面试

But over the years, countless students of mine have regaled me with stories about how mastering the concepts in this class enabled them to ace every technical question they were ever asked.

但多年以来，我数不尽的学生给我讲述了精通这门课程上的概念是如何帮助他们解决每一个他们被问到的技术问题

I told you, this is fundamental stuff. So, what do I expect from you? Well, honestly, the answer is nothing. After all isn't the whole point of a free online class like this one that anyone can take it and devote as much effort to it as they like.

我告诉过你，这是基础的东西，所以，我对你有什么期待吗？好吧，老实说，答案是什么都没有。毕竟，像这样一个免费的在线课程，不是所有人都可以接受，并尽其可能的投入到它上面。

So that said, as a teacher it's still useful to have one or more canonical students in mind. And I thought I'd go ahead and be transparent with you about how I'm thinking about these lectures.

所以说，作为一名老师，有一个或多个规范学生在脑子里是仍然有用的。我想我应该继续前进，向你们坦率我对这些讲座是如何想的。

Who I have in mind that I'm teaching to. So again, please don't feel discouraged if you don't conform to this canonical student template. I'm happy to have the opportunity to teach you about algorithms no matter who you are.

我会记住我在教谁，所以，如果你不符合这个规范学生的模板，请不要感到气馁，我很高兴有机会能够教你算法，不管你是谁

So first, I have in mind someone who knows at least some programming. For example, consider the previous lecture. We talked about a recursive approach to multiplying two numbers and I mentioned how in certain mathematical expression, back then we labeled it star and circled it in green.

所以首先，我会记住那些至少知道一些编程的人。举个例子，细想前一节课，我们谈论一个递归的途径去计算两个数的乘法，而且，我提到了在某些数学表达式中，我们将它标记为星形，并且用绿色圈起来

How that expression naturally translated into a recursive algorithm. In particular, I was certainly assuming that you had some familiarity with recursive programs.

该表达式如何自然的转换成一个递归算法。特别是，我当时必定假设了你对递归程序有一定的了解

If you feel comfortable with my statement in that lecture, if you feel like you could code up a recursive integer multiplication algorithm based on the high level outline that I gave you, then you should be in good shape for this course. You should be good to go.

如果你对我在讲座中的陈述感到舒适，如果你觉得你可以编写一个递归整数乘法，给予我给你的高级大纲，那么，你在这门课中应该会有良好的状态，你应该好好去(学)

If you weren't comfortable with that statement, well, you might not be comfortable with the relatively high conceptual level at which we discuss program in this course. But I encourage to watch the next several videos anyway, to see if you get enough out of them to make it worth your while.

如果你对那些陈述没有感到舒适，好吧，你也许会对我们在这节课中讨论的相对较高的概念水平感到不适，但是，我鼓励你无论如何看完接下来的几个视频，去看你是否能从中获得足够多的东西，让它值得你花时间。

Now, while I'm aiming these lectures at people who know some programming, I'm not making any assumptions whatsoever about exactly which programming languages you know.

现在，虽然我把这些讲座对准那些知道一些编程的人，但我并没有对你知道那些编程做任何假设

Any standard imperative language you know, something like C, Java or Python, is totally fine for this course. Now, to make these lectures accessible to as many

programmers as possible, and to be honest, you know, also to promote {thinking about programming} at a relatively abstract conceptual level, I won't be describing algorithms in any particular programming language.

你知道的任何标准命令式语言，如C，Java或Python，对于本课程来说都是完全可以的。 现在，为了让尽可能多的程序员可以访问这些讲座，并且说实话，你也知道，为了促进在相对抽象的概念层面上编程的思考，我不会用任何特定的编程语言来描述算法。

Rather, when I discuss the algorithms, I'll use only high-level pseudo-code, or often simply English. My inductive hypothesis is that you are capable of translating such a high level description into a working program in your favorite programming language.

相反，当我讨论算法是，我将只会使用高级伪代码，或者经常是简单的英语。我的归纳假设是你有能力用你最喜欢的编程语言将一个高级描述转化成一个可工作的程序。

In fact, I strongly encourage everyone watching these lectures to do such a translation of all of the algorithms that we discussed. This will ensure your comprehension, and appreciation of them. Indeed, many professional computer scientists and programmers don't feel that they really understand an algorithm until they've coded it up.

事实上，我强烈建议每个人去看那些将我们讨论的算法做如此转化的讲座。这将确保你理解并且欣赏它们。实际上，很多专业的计算机科学家和程序员不觉得他们真的理解一个算法知道他们将这个算法写出来

Many of the course's assignments will have a problem in which we ask you to do precisely this. Put another way, if you're looking for a sort of coding cookbook, code that you can copy and paste directly into your own programs.

很多课程作业会有一个问题，恰恰是我们让你做的。换句话说，如果你正在寻找一种编码菜谱，那些你可以立即复制粘贴到你自己的程序中的代码

Without necessarily understanding how it works, then this is definitely not the course for you. There are several books out there that cater to programmers looking for such coding cook books.

而不是理解它如何工作，那么这个课程一定不是你的课程。有好几本书是为了迎合寻找这些编码菜谱的程序员的。

Second, for these lectures I have in mind someone who has at least a modest amount of mathematical experience though perhaps with a fair bit of accumulated rust.

第二，在这些讲座中，我会记住那些至少有适量的数学经验的人，尽管也许有不少锈迹累积

Concretely I expect you to be able to recognize a logical argument that is a proof. In addition, two methods of proof that I hope you've seen before are proofs by induction and proofs by contradiction.

具体的说，我希望你能认出一个证明的逻辑推理。另外，我希望你以前见过归纳证明和矛盾证明这两种证明方法

I also need you to be familiar with basic mathematical notation, like the standard quantifier and summation symbols.

我也需要你对基础的数学符号熟悉，比如标准量词和求和符号

A few of the lectures on randomized algorithms and hashing will go down much easier for you if you've seen discrete probability at some point in your life.

如果你曾经看过离散概率，那么随机算法和散列的一些讲座你将会更加容易学会

But beyond these basics, the lectures will be self contained. You don't even need to know any calculus, save for a single simple integral that magically pops up in the analys of the randomized quick sort algorithm.

但除开这些基础以外，讲座将自成一体，除了在随机快速排序算法分析中神奇弹出的单个简单积分外，你甚至不需要知道任何微积分

I imagine that many of you have studied math in the past, but you could use a refresher, you're a bit rusty. And there's plenty of free resources out there on the web, and I encourage you to explore and find some that you like. But one that I want to particularly recommend is a great set of free lecture notes.

我想你们中的大多数都曾经学习过数学，但是你可以使用一个补习课程，你有点生疏。网上有大量的免费资源，而我鼓励你去探索和寻找其中你喜欢的。但是我想特别推荐的是一套很棒的免费教学讲义

It's called Mathematics for Computer Science. It's authored by Eric Lehman and Tom Layden, and it's quite easy to find on the web if you just do a web search.

它叫《Mathematics for Computer Science》。它的作者是Eric Lehman 和 Tom Layden，它在网上特别容易找到，只要你做一次网页搜索

And those notes cover all of the prerequisites that we'll need, in addition to tons of other stuff. In the spirit of keeping this course as widely accessible as possible, we're keeping the required supporting materials to an absolute minimum.

这些笔记涵盖了所有我们所需要的先决条件，以及大量其他东西。本着保证这个课程尽可能广泛的可入门。我们将必须的支持材料保持在了最低限度

Lectures are meant to be self-contained and we'll always provide you with the lecture notes in PowerPoint and PDF format. Once in a while, we'll also provide some additional lecture notes. No textbook is required for this class. But that said, most of the material that we'll study is well covered in a number of excellent algorithms books that are out there. So I'll single out four such books here.

讲座打算要自成一体，同时，我们将始终将讲座笔记以 PPT 和 PDF 格式提供给你。偶尔，我们也会提供一些额外的讲座笔记。本课程不需要课本。但是，即便如此，大部分我们将要学到的材料都很好的包含在了许多出色的外部算法书忠。所以，我会在这里挑出四本这样的书

The first three I mention because they all had a significant influence on the way that I both think about and teach algorithms. So it's natural to acknowledge that debt here.

我提到的前三个，因为他们对我的思考和教授算法都有很深远的影响，所以在这个表达对这个恩情的感谢；

One very cool thing about the second book, the one by Dasgupta, Papadimitriou and Vazirani, is that the authors have made a version of it available online for free.

关于第二本书很酷的一件事情是，它的作者，Dasgupta、Papadimitriou 和 Vazirani，在网上提供了一个免费版本。

And again, if you search on the authors' names and the textbook title, you should have no trouble coming up with it with a web search. Similarly, that's the reason I've listed the fourth book because those authors have likewise made essentially a complete version of that book available online and it's a good match for the material that we're going to cover here.

再次，如果你在搜索引擎中搜索作者的名字和书名，你会毫不费力的获得它。同样的，这也是我列出第四本书的原因，因为它的作者们也同样已经基本上在网上提供了该书的完整版本，并且它与我们在这里将要介绍的材料相配

If you're looking for more details about something covered in this class, or simply a different explanation than the one that I give you, all of these books are gonna be good resources for you.

如果你正在寻找这节课介绍的一些东西的更多细节，或者简单的寻找一个和我给你的那个不一样的解释，所有这些书对你来说会是好的资源

There are also a number of excellent algorithm textbooks that I haven't put on this list. I encourage to explore and find you own favorite.

还有很多出色的算法课本我不能放在这个列表中，我鼓励去探索寻找你自己喜欢的。

In our assignments, we'll sometimes ask you to code up an algorithm and use it to solve a concrete problem that is too large to solve by hand. Now, we don't care what program and language and development environment you use to do this as we're only going to be asking you for the final answer.

在我们的作业中，我们有时会要求你对一个算法编程，并且用它去解决一个手动解决很复杂的具体的问题。现在，我们不在意你用什么程序、语言、开发环境去做这个，因为我们只要求你给出最终答案。

Thus, we're not requiring anything specific, just that you are able to write and execute programs. If you need help or advice about how to get set up with a suitable coding environment, we suggest that you ask other students for help via the course discussion forum.

因此，我们不需要任何特定的东西，只需要你能够编写和执行程序。如果你需要有关如何使用合适的编码环境进行设置的帮助或建议，我们建议你可以通过课程讨论论坛去寻求其他学生的帮助。

Finally, let's talk a bit more about assessment. Now this course doesn't have official grades per se, but we will be assigning weekly homeworks. Now we're going to assign homeworks for three different reasons.

最后，让我们再谈谈评定。现在这门课本身没有官方的成绩，但是我们将分配每周的家庭作业。现在，我们有三个不同的原因去分配家庭作业。

The first is just for self-assessment. It's to give you the opportunity to test your understanding of the material so that you can figure out which topics you've mastered and which ones that you haven't.

第一个原因是为了自我评定。它给你机会去测试你对材料的理解，以便你可以确定你掌握了哪些主题，又有哪些你没有掌握

The second reason we do it is to impose some structure on the course, including deadlines, to provide you with some additional motivation to work through all the topics. Deadlines also have a very important side effect that synchronizes a lot of the students in the class. And this of course makes the course discussion forum a far more effective tool for students to seek and provide help in understanding the course material.

我们这样做的第二个理由是在课程上强加一些结构，包括截止日期，以来给你提供一些额外的动力去完成所有的主题。截止日期还有一个非常重要的副作用，那就是能同步课堂上的很多学生。这当然会使课程讨论论坛成为学生在理解课程材料时寻求和提供帮助的更有效的工具

The final reason that we give homeworks is to satisfy those of you who, on top of learning the course material, are looking to challenge yourself intellectually.

我们提供作业的最后一个原因是为了满足那些熟练掌握课程所学的材料，寻求自我智力挑战的人。

Now, this class has tens of thousands of students. So it's obviously essential that the assignments can be graded automatically. Now, we're currently only in the 1.0 generation of free online courses such as this one. So the available tools for auto graded assessment are currently rather primitive.

现在，这门课有成千上万的学生，所以一个可以自动评分的分配显然是必要的。现在像这样的免费课程我们还在 1.0代。所以，用于自动评分评估的可用工具现在还相当原始

So, we'll do the best we can, but I have to be honest with you. It's difficult, or maybe even impossible to test deep understanding of the design and analysis of algorithms, using the current set of tools.

所以，我们会尽我们所能，但我必须对你坦诚的是，使用当前的工具集来测试对算法的设计和分析的深度理解是很困难，甚至是不可能的

Thus, while the lecture content in this online course is in no way watered down from the original Stanford version. The required assignments and exams we'll give you, are not as demanding as those that are given in the on campus version of the course.

因此，虽然这门在线课程中的讲座内容绝不会比原始斯坦福版本有稀释，但是我们将会给你需要的作业和考试，并不会像在校园里给你的课程版本那么严格

To make up for this fact, we'll occasionally propose optional algorithm design problems, either in a video or via supplementary assignment. We don't have the ability to grade these, but we hope that you'll find them interesting and challenging, and that you'll discuss possible solutions with other students via the course discussion forum.

为了弥补这个事实，我们将会偶尔在视频或者通过追加的分配提出可选的算法设计问题，。我们没有能力去对这些评分，但我们希望你可以找到其中的有趣和挑战，同时，你将可以和其他学生在课程论坛上讨论可能的解决方案。

So I hope this discussion answered most of the questions you have about the course. Lets move on to the real reason that we're all here, to learn more about algorithms.

所以我希望这样的讨论可以解决你在课程中遇到的大部分问题。让我们前进到我们都在这里的真实原因：去学习更多关于算法的东西。

**Merge Sort：Motivation and Example**

Okay. So in this video, we'll get our first sense of {what it's actually like} to analyze an algorithm. And we'll do that {by first of all} reviewing a famous sorting algorithm, namely the Merge Sort algorithm.

好，在这个视频中，我们将第一次了解分析算法的实际情况。我们将首先回顾一个著名的排序算法，叫做归并排序算法。

And then giving a really fairly mathematically precise upper bound on exactly how many operations the Merge Sort algorithm requires to correctly sort an input array.

然后对归并排序算法正确的将一个输入数组排序需要多少操作次数给出一个相当精确的数学上界

So I feel like I should begin with a bit of an apology. Here we are in 2012, a very futuristic sounding date. And yet I'm beginning with a really quite ancient algorithm. So for example, Merge Sort was certainly known, to John Von Neumann all the way back in 1945.

所以，我想我应该先从一个小小的道歉开始。现在是 2012 年，一个非常前卫的时间。但是我却懂一个是在非常古老的算法开始讲。举个例子，对约翰·冯·诺依曼来说，归并排序早在 1945 年就已经众所周知了

So, what justification do I have for beginning, you know, a modern class in algorithms with such an old example? Well, there's a bunch of reasons.

所以，是什么理由让我从一个，你知道的，一个现代的算法课程里讲这样一个老例子呢？好吧，有一堆理由。

One, I haven't even put down on the slide, which is like a number of the algorithms we'll see, "Merge Sort" as an oldie but a goodie.

第一个原因，我甚至没有把它记载幻灯片里，这个原因就像我们将会看到的若干算法一样。“归并排序”虽然是个老人，但是个好人

So it's over 60, or maybe even 70 years old. But it's still used all the time in practice, because this really is one of the methods of choice for sorting. The standard sorting algorithm in the number of programming libraries. So that's the first reason.

所以，尽管它超过了 60，甚至也许超过了 70 岁，但它仍然在实践中一直被使用，因为这个算法确实是排序方式的选择之一。是很多编程库的标准排序算法。所以，这是第一个原因

But there's a number of others as well that I want to be explicit about. So first of all, throughout these online courses, we'll see a number of general algorithm design paradigms ways of solving problems that cut across different application domains.

但是我还想明确其他一些事情，首先，在这个在线课程中，我们将看到很多通用算法设计范例，以解决跨越不同应用领域的问题

And the first one we're going to focus on is called the Divide-and-Conquer algorithm design paradigm. So in Divide-and-Conquer, the idea is, you take a problem, and break it down into smaller sub problems which you then solve recursively, ...... and then you somehow combine the results of the smaller sub-problem to get a solution to the original problem that you actually care about.

我们将聚焦的第一个范例叫做分治算法设计范例。在分治算法中，方法是，你拿到了一个问题，然后将它分解成更小的代替问题，然后你递归解决，然后，你用某种方式将这些更小的代替问题的答案组合起来，从而获得你实际关注的原始问题的解决方案。

And Merge Sort is still today's the, perhaps the, most transparent application of the Divide-and-Conquer paradigm, ...... that will exhibit very clear what the paradigm is, what analysis and challenge it presents, and what kind of benefits you might derive.

归并排序算法也许仍然是今天分治范式最透明的应用，它会很清晰的展示出这个范式是什么，它提出了什么样的分析和挑战，以及你可能获得什么样的好处

As for its benefits, so for example, you're probably all aware of the sorting problem. Probably you know some number of sorting algorithms perhaps including Merge Sort itself.

至于它的好处，举个例子，你可能都知道排序问题，可能你知道一些排序算法也许包括归并排序本身。

And Merge Sort is better than a lot of this sort of simpler, I would say obvious, sorting algorithms, ...... so for example, three other sorting algorithms that you may know about, but that I'm not going to discuss here.

归并排序比很多这些更简单的排序要更好，我会说很明显，排序算法，所以，举个例子，也许你知道的其他三个排序算法，但是我不打算在这里讨论

If you don't know them, I encourage you to look them up in a text book or look them up on the web.Let's start with three sorting algorithms which are perhaps simpler, first of all is "Selection Sort".

如果你不知道他们，我鼓励你在课本或者网上找找他们。让我们从三个排序算法开始，它们也许更简单，首先是是选择排序。

So this is where you do a number of passes through the way repeatedly, identifying the minimum of the elements that you haven't looked at yet, ......so you're basically a linear number of passes each time doing a minimum computation.

所以你通过选择排序重复的做大量的传递，识别你还没有看到的最小元素，所以你基本上每次做最小数计算都是一个线性数的传递；

There's "Insertion Sort", which is still useful in certain cases in practice as we will discuss, but again it's generally not as good as Merge Sort, ...... where you will repeatedly maintain the invariant that prefix view of array, which is sorted version of those elements.

插入排序，我们将讨论到它在某些情况下仍然有用，但是通常不如归并排序。其中，你讲反复的维护数组的前缀视图的不变量，这是这些元素的排序版本

So after ten loops of Insertion Sort, you'll have the invariant that whatever the first ten elements of the array are going to be in sorted order, ...... and then when Insertion Sort completes, you'll have an entire sorted array.

所以插入排序在十次循环之后，你将会获得不变数，即数组前十个元素将按排序顺序顺序排列，然后，当插入排序完成后，你将会获得完整的已排序的数组

Finally, some of you may know about "Bubble Sort", which is where you identify adjacent pairs of elements which are out of order, ... and then you do repeated swaps until in the end the array is completely sorted.

最后，你们其中有些人应该听过布隆排序，这个排序会识别邻近的一对乱序元素，然后你将会重复交换，知道最后数组完成排序。

Again I just say this to jog your memory, these are simpler sorts than Merge Sort, ...... but all of them are worse in the sense that they're lack in performance in general, which scales with N^2, ...... and the input array has N elements, so they all have, in some sense, quadratic running time.

再次，我说这些只是唤起你的记忆，这些排序比归并排序更简单，但这些都更糟糕，因为它们一般缺乏性能，它与N^2成比例，并且输入的数组有 N个元素，所以在某种意义上它们都具有二次的运行时间

But if we use this non-trivial Divide-and-Conquer approach, or non-obvious approach, we'll get a, as we'll see, a much better running time than this quadratic dependence on the input.

但是如果我们使用这个非凡的分治方法或者非显而易见的方法，我们将获得一个，就像我们看到的，一个比这个取决于输入数量的二次方更好的运行时间

Okay? So we'll get a win, first sorting in Divide-and-Conquer, and Merge Sort is the algorithm that realizes that benefit.

好的？我们将获得胜利，首先在分治范式中排序，归并排序是实现这种好处的算法

So the second reason that I wanna start out by talking about the Merge Sort algorithm, is to help you calibrate your preparation. I think the discussion we're about to have will give you a good signal for whether you're background's at about the right level, of the audience that I'm thinking about for this course.

所以我想从归并排序开始讲述来给你一个帮助去校正你的准备。我认为我们将要讨论的东西会给你一个很好的信号：你是否具有适当水平的背景，来作为我正在为这门课程考虑的观众

So in particular, when I describe the Merge Sort algorithm, you'll notice that I'm not going to describe in a level of detail that you can just translate it line by line into a working program in some programming language.

特别的，当我描述归并排序算法时，你将会注意到，我不会在一个细节层面描述，那就是你只是可以使用一些编程语言一行一行的将它翻译成一个可运行的程序

My assumption again is that you're a sort of the programmer, and you can take the high-level idea of the algorithm, how it works, ...... and you're perfectly capable of turning that into a working program in whatever language you see fit.

我的假设是你是一个程序员，你可以获得算法的高级思想，它如何工作…… 同时，你能完美胜任，用任何你看着舒服的语言，将它转换成一个可运行的程序

So hopefully, I don't know, it may not be easy the analysis of Merge Sort discussion. But I hope that you find it at least relatively straight forward, .... because as the course moves on, we're going to be discussing algorithms and analysis which are a bit more complicated than the one we're about to do with Merge Sort.

所以希望，我不知道，也许对于归并算法分析的讨论不会很简单，但是我希望你至少可以相对直截了当的发现它，因为随着课程向前推进，我们将会讨论那些比我们马上要做的归并排序还要复杂一些的算法和分析

So in other words, I think that this would be a good warm-up for what's to come. Now another reason I want to discuss Merge Sort is that our analysis of it will naturally segment discussion of how we analyze the algorithms in this course and in general.

所以换句话说，我认为这对即将到来的东西是一个很好的热身。我另外一个想要讨论归并排序算法的理由是，我们对归并排序的分析将自然地分割关于我们通常如何在这节课中分析算法的讨论。

So we're going to expose a couple of assumptions in our analysis, we're focus on worst case behavior, ...... or we'll look for guarantees on performance on running time that hold for every possible input on a given size,

所以我们将会在我们的分析中揭示一些假设，我们将关注最坏情况的行为，或者我们将寻找适用于每个可能的给定长度的输入的运行时间的性能保证

... and then we'll also {expose our focus} on so called "Asymptotic Analysis", which meaning will be much more concerned with the rate of growth on an algorithms performance than on things like low-order terms or on small changes in the constant factors.

然后，我们也将把我们的重点放在所谓的“渐进分析“，这种手段将会更关注一个算法性能的增长率，而不是考虑低阶项或者常数因子的小改变

Finally, we'll do the analysis of Merge Sort using what's called as "Recursion-Tree" method. So this is a way of tying up the total number of operations that are executed by an algorithm.

最后，我们将适用被称为“递归树”的方式去分析归并排序，这是一种将一个算法执行操作总数捆绑起来的一种方法

And as we'll see a little bit later, this Recursion-Tree method generalizes greatly. And it will allow us to analyze lots of different recursive algorithms, lots of different Divide-and-Conquer algorithms, including the integer multiplication algorithm that we discussed in an earlier segment.

正如我们稍后将会看到的那样，这个递归树方法非常通用。它将允许我们分析很多不同的递归算法，很多不同的分治算法，包括我们在前面一节讨论的整数乘法算法

So those are the reasons to start out with Merge Sort. So what is the computational problem that Merge Sort is meant to solve? Well, presumably, you all know about the sorting problem. But let me tell you a little bit about it anyways, just so that we're all on the same page.

所以，这些就是从归并排序开始的原因。所以归并排序打算解决什么计算问题呢？好吧，大概，你们都知道排序问题，但是无论如何，让我告诉你们一点，只是为了让我们都在同一页上

So, we're given as input. An array of N numbers in arbitrary order, and the goal of course is to produce output array where the numbers are in sorted order, let's say, from smallest to largest.

所以我们作为输入。一个N 个数字的数组，仍以顺序，当然目标是产生输出数组，其中数字按照顺序排序，比如说，从最小到最大

Okay so, for example, we could consider the following input array, and then the goal would be to produce the following output array. Now one quick comment. You'll notice that here in input array, it had eight elements, all of them were distinct, it was the different integers, between 1 and 8.

好，所以，举个例子，我们会考虑下面的输入数组，然后目标是生产下面的输出数组，现在，一个快速的描述。你将会注意到这里的输入数组，他有八个元素，所有元素都是不重复的，它们是不同的整数，从1 到 8

Now the sorting problem really isn't any harder if you have duplicates, in fact it can even be easier, ...... but to keep the discussion as simple as possible let's just, among friends, go ahead and assume that they're distinct, for the purpose of this lecture.

现在，这个排序问题真的没有任何难度，如果你有副本的话，事实上他甚至可以更简单。但是为了尽可能简单的讨论，让我们在朋友之中进行，并假设它们是不同的，为了本次讲座的目的

And I'll leave it as an exercise which I encourage you to do, which is to think about how the Merge Sort algorithm implementation and analysis would be different, if at all, if there were ties, okay?

我将把它作为一个练习，我鼓励你去做，去思考归并排序的实现和分析有什么不同，如果有联系的话，好吗？

Go ahead and make the distinct assumption for simplicity from here on out. Okay, so before I write down any pseudo code for Merge Sort, let me just show you how the algorithm works using a picture, ...... and I think it'll be pretty clear what the code would be, even just given a single example.

从现在开始，并且做出一个简单的假设。好的，所以在我写下任何归并排序的伪代码之前，让我先使用一张图片告诉你算法是如何工作的，我认为代码会非常清楚，尽管只是提供了一个简单地例子

So let's go ahead and consider the same unsorted input array that we had on the previous slide. So the Merge Sort algorithm is a recursive algorithm, and again, that means that a program which calls itself and it calls itself on smaller sub problems of the same form, okay?

所以让我们继续考虑与我们在上一张幻灯片相同的未排序的输入数组。所以，归并排序算法是一个递归算法，以及，这意味着，一个问题会被它自己调用或者它会通过相同的形式通过一个较小的子问题来调用它自己，好吗？

So the Merge Sort is its purpose in life is to sort the given input array. So it's going to spawn, or call itself on smaller arrays. And this is gonna be a canonical Divide-and-Conquer application, where we simply take the input array, we split it in half, we solve the left half recursively, we solve the right half recursively, and then we combine the results.

因此，归并排序的目的是将给定的输入数组排序，所以它会产生，或者通过更小的数组调用它自己，这就是一个经典 的分治应用，我们简单的给一个输入数据，我们将它分割成两半，我们递归解决左半部分，我们递归解决有半部分，然后，我们将结果组合起来。

So let's look at that in the picture. So the first recursive call gets the first four elements, the left half of the array, namely 5, 4, 1, 8. And, of course, the other recursive call is gonna get the rest of the elements, 7, 2, 6, 3. You can imagine these has been copied into new arrays before they're given to the recursive calls.

那么，让我们来看一下图片中的内容，第一次递归调用获得前四个元素，数组的左半部分，即 5，4，1，8，当然，另外的一次递归调用会获得剩下的元素，即 7，2，6，3，你可以想象一下它们在被给到递归调用之前被拷贝成两个新的数组。

Now, by the magic of recursion, or by induction if you like, the recursive calls will do their task. They will correctly sort each of these arrays of four elements, and we'll get back sorted versions of them.

现在，通过递归的魔力，或者通过归纳法，如果你喜欢的话，递归调用将会完成它们的任务。它们将会正确的对这些四个元素的数组排序，并且我们将取回它们排过序的版本。

So from our first recursive call, we receive the output, 1, 4, 5, 8, and from the second recursive call, we received the sorted output, 2, 3, 6, 7. So now, all the remains to complete the Merge Sort is to take the two results of our recursive calls, these two sorted elements of length-4, and combine them to produce the final output, namely the sorted array of all eight of the input numbers.

所以从我们第一次递归调用，我们接收到了输出，1，4，5，8，然后从第二次递归调用，我们接受了排序后的输出2，3，6，7，所以现在，所有完成归并排序后的剩余就是取得两次递归调用的结果，这两个长度为 4 的有序元素，将它们组装起来生产最后的输出，也就是说，所有八个输入的数字的排序数组

And this is the step which is called "Merge". And hopefully you are already are thinking about how you might actually implement this merge in a computationally efficient way. But I do owe you some more details. And I will tell you exactly how the merge is done.

这就是称之为“归并”的步骤。希望你已经在思考如何用有效的计算方式来真正实现这个合并。但是我应该给予你更多细节，我将会准确的告诉你，合并是如何完成的

In effect, you just walk pointers down each of the two sort of sub-arrays, copying over, populating the output array in the sorted order. But I will give you some more details in just a slide or two. So that's Merge Sort in a picture. Split it in half, solve recursively, and then have some slick merging procedure to combine the two results into a sorted output.

事实上，你只需要沿着这两种子数组的每一种向下移动指针，复制，按排序顺序填充输出数组。但我将会用一两张幻灯片给你更多的一些细节，这就是图片中的归并排序。将它切割成两半，递归解决，然后使用一些灵活的排序步骤将两个结果组合成一个顺序的输出

**Merge Sort: Pseudocode**

Okay, so let's move on, and actually discuss the pseudo-code for the merge sort algorithm. First, let me just tell you the pseudo-code, leaving aside exactly how the merging subroutine is implemented. And thus, high levels should be very simple and clear at this point.

好，让我们继续，真正的讨论归并排序算法的伪代码。首先，让我只是告诉你伪代码，不谈合并子程序是如何运行的。因此，在这一点上，高等级应该会非常简单和清晰

So there's gonna be two recursive calls, and then there's gonna be a merging step. Now, I owe you a few comments, 'cause I'm being a little sloppy. Again, as I promised, this isn't something you would directly translate into code, although it's pretty close.

这将有两次递归调用，然后将要一步合并。现在，我应该给你一些建议，因为我有点粗心，再次，就像我承诺的那样，这不是你可以立即翻译成代码的东西，尽管它非常接近。

But so what are the couple of the ways that I'm being sloppy? Well, first of all, there's, [inaudible], you know, in any recursive algorithm, you gotta have some base cases. You gotta have this idea that when the input's sufficient.

但是，那我变得粗心是哪两种方式呢？好吧，首先，你知道，在任何递归算法种，你必须有一些基础实例。你必须有这个想法，当输入足够时

Really small you don't do any recursion, you just return some trivial answer. So in the sorting problem the base case would be if your handed an array that has either zero or an elements, well it's already sorted, there's nothing to do, so you just return it without any recursion.

足够小到你不用做任何递归，你只需要返回一些不重要的答案。所以在排序问题种，基础实例就是：如果你有一个数组，有两个零或者一个元素，它已经排过序，已经没有事情可做，所以你只需要返回它，而不需要任何递归。

Okay, so to be clear, I haven't written down the base cases. Although of course you would if you were actually implementing, a merge short. Some of you, make a note of that.

好吧，所以说清楚，我还没有写下基础实例，尽管你当然会实现，如果你曾经真正实现了一个合并短代码。你们中的一些人，做笔记。

A couple of other things I'm ignoring. I'm ignoring what the, what to do if the array has odd lengths, so if it has say nine elements, obviously you have to somehow break that into five and four or four and five, so you would do that just in either way and that would fine.

还有一些我忽略的事情，我忽略了什么，如果数组是基数长度该怎么办，如果有 9 个 元素，很明显你应该用某种方法将它们拆分成 5 和 4 或者 4 和 5，所以你会以任何一种方式做到，这很好。

And then secondly, I'm ignoring the details or what it really means to sort of recursively sort, so for example, I'm not discussing exactly how you would pass these subarrays onto the recursive calls.

然后，第二，我忽略了细节或它对递归排序的真正意义，举个例子，我没有真正讨论你如果将这些子数组传递给递归调用

That's something that would really depend somewhat on what, on the programming language, so that's exactly what I want to avoid. I really want to talk about the concepts which transcend any particular programming language implementation. So that's why I'm going to describe algorithms at this level okay.

这在一定程度上，取决于编程语言。这正是我想避免的，我真正想谈谈超越任何具体编程语言实现的概念。所以，那就是为什么我要在这个层次描述算法

Alright, so the hard part relatively speaking, that is. How do you implement the merge depth? The recursive calls have done their work. We have these two sort of separated half the numbers. The left half and the right half. How do we combine them into one?

好吧，所以相对难讲的部分，是如果实现合并深度？递归调用已经完成了它们的工作，我们有这两种数字分开一般的数字。左半部分和有半部分，我们如何将它们组成一个呢？

And in English, I already told you on the last slide. The idea is you just populate the output array in a sorted order, by traversing pointers or just traversing through the two, sorted sub-arrays in parallel.

我已经在最后一张幻灯片告诉你了，思想是你只需要按顺序填充输出数组，通过指针遍历或者并行便利两个已排序的子数组

So let's look at that in some more detail. Okay, so here is the pseudo-code for the merge step. [sound] So let me begin by, introducing some names for the, characters in the, what we're about to discuss. So let's use C. To denote the output array.

让我们更详细的看一下。好，这里是合并步骤的伪代码。让我我首先介绍一下我们即将讨论的角色名称。让我们用 C，来表示输出数组。

So this is what we're suppose to spit out with the numbers in sorted order. And then, I'm gonna use a and b to denote the results of the two recursive calls, okay? So, the first recursive call has given us array a, which contains the left half of the input array in sorted order. Similarly, b contains the right half of the input array, again, in sorted order.

所以这是我们认为吐出按顺序排列的数字。然后，我将使用a 和 b 去表示两个递归调用的结果，好吗？所以，第一次低估调用给我们数组 a，它包含了按顺序排列的左半部分输入数组。同样的，b 包含了有半部分的输入数据，同样，也是按顺序排列

So, as I said, we're gonna need to traverse the two, sorted sub-arrays, a and b, in parallel. So, I'm gonna introduce a counter, i, to traverse through a, j to traverse through b. I and j will both be initialized to one, to be at the beginning of their respective arrays.

所以，就像我说的，我们将需要并行遍历两个已排序的子数组，a 和b，所以，我将要介绍一个计数器：i去遍历a，j 去遍历 b。I 和 j 将都初始化为一个，位于它们各自数组开头。

And now we're gonna do. We're going to do a single pass of the output array copying it in an increasing order. Always taking the smallest from the union of the two sorted sub arrays.

现在我么将去做，我们将要对输出数据进行单次传递，以递增的顺序复制它。总是从两个已排序的子数组的并集中取出最小值

And if you, if there's one idea in this merge step it's just the realization that. The minimum element that you haven't yet looked at in A and B has to be at the front of one or the two lists right so for example at the very beginning of the algorithm where is the minimum element over all. Well, which ever of the two arrays it lands in -- A or B -- it has to be the smallest one there okay.

如果你，如果在这个合并步骤中有一个想法，那就肯定意识到了。你在 A 和 B 中尚未看到的最小值，必须在一个或两个列表的最前面，例如在算法的最开始处，其中是全部元素的最小元素，那么，它所在的数组，A 或者 B，它肯定有最小的数

So the smallest element over all is either the smallest element A or it's the smallest element B. So you just check both places, the smaller one is the smallest you copy it over and you repeat. That's it. So the purpose of K is just to traverse the output array from left to right.

所以全部元素的最小元素要不是元素 A 的最小元素，要不是元素 B的最小元素。所以你只需要检查这两个地方，更小的那个就是最小的，你复制它然后重复，而已，所以KDE 目标只是遍历输出数组，从左往右

That's the order we're gonna populate it. Currently looking at position I, and the first array of position J and the second array. So that's how far we've gotten, how deeply we've probed in the both of those two arrays. We look at which one has the current smallest, and we copy the smallest one over. Okay?

这就是我们要填充它的顺序，现在查看I的位置在第一数组中，以及 J 的位置，在第二数组中，这就是我们已经走了多远，我们在这两个数组中探测多深。我们查看哪一个有目前最下值，然后我们拷贝最小的那个。好吗？

So if the, if, the entry in the i position of A is smaller, we copy that one over. Of course, we have to increment i. We probe one deeper into the list A, and symmetrically for the case where the current position in B has the smaller element.

因此，如果在 A中的位置上的条目更小，我们拷贝这个值。当然，我们要增加 i，我们在 A 列表中探索更深的一个，也要在 B 当前位置有更小的元素的情况下，对称的探索。

Now again, I'm being a little bit sloppy, so that we can focus on the forest, and not sort of, And not get

bogged down with the trees. I'm ignoring some end cases, so if you really wanted to

implement this, you'd have to add a little bit, to keep track of when you fall off,

either, either A or B. Because you have additional checks for when i or j reaches

the end of the array, at which point you copy over all the remaining elements into

C. Alright, so I'm gonna give you a cleaned up version, of, that pseudo-code

so that you don't have to tolerate my questionable handwriting any longer than

is absolutely necessary. This again, is just the same thing that we wrote on the

last slide, okay? The pseudo-code for the merge step. Now, so that's the Merge Sort

algorithm. Now let's get to the meaty part of this lecture, which is, okay, so merge

sort produces a sorted array. What makes it, if anything, better than much simpler

non divide and conquer algorithms, like say, insertion sort? Other words, what is

the running time of the merge sort algorithm? Now I'm not gonna give you a

completely precise definition, definition of what I mean by running time and there's

good reason for that, as we'll discuss shortly. But intuitively, you should think

of the running time of an algorithm, you should imagine that you're just running

the algorithm in a debugger. Then, every time you press enter, you advance with one

line of the program through the debugger. And then basically, the running time is

just a number of operations executed, the number of lines of code executed. So the

question is, how many times you have to hit enter on the debugger before the,

program finally terminates. So we're interested in how many such, lines of code

get executed for Merge Short when an input array has n numbers. Okay, so

that's a fairly complicated question. So let's start with a more modest school.

Rather than thinking about the number of operations executed by Merge Sort, which

is this crazy recursive algorithm, which is calling itself over and over and over

again. Let's just think about how many operations are gonna get executed when we

do a single merge of two sorted sub arrays. That seems like it should be an

easier place to start. So let me remind you, the pseudo code of the merge

subroutine, here it is. So let's just go and count up how many operations

that are gonna get used. So there's the initialization step. So let's say that

I'm gonna charge us one operation for each of these two initializations. So let's

call this two operations, just set i equal to one and j equal to one then we have this four

loop executes a total number of end times so each of these in iterations of this

four loop how many instructions get executed, well we have one here we have a

comparison so we compare A(i) to B(j) and either way the comparison comes up we then

do two more operations, we do an assignment. Here or here. And then we do

an increment of the relevent variable either here or here. So that's gonna be

three operations per iteration. And then maybe I'll also say that in order to

increment K we're gonna call it a fourth iteration. Okay? So for each of these N

iterations of the four loop we're gonna do four operations. All right? So putting it

all together, what do we have is the running time for merge. So let's see the

upshot. So the upshot is that the running time of the merge subroutine, given an

array of M numbers, is at most four M plus two. So a couple of comments. First of

all, I've changed a letter on you so don't get confused. In the previous slide we

were thinking about an input size of N. Here I've just made it. See I've changed

the name of the variable to M. That's gonna be convenient once we think about

merge sort, which is recursing on smaller sub-problems. But it's exactly the same

thing and, and whatever. So an array of M entries does as most four M plus two.

Lines of code. The second thing is, there's some ambiguity in exactly how we

counted lines of code on the previous slide. So maybe you might argue that, you

know, really, each loop iteration should count as two operations, not just

one.'Cause you don't just have to increment K, but you also have to compare

it to the, upper bound of N. Eh, maybe. Would have been 5M+2 instead of 4M+2. So

it turns out these small differences in how you count up. The number of lines of

code executed are not gonna matter, and we'll see why shortly. So, amongst

friends, let's just agree, let's call it 4M plus two operations from merge, to

execute on array on exactly M entries. So, let me abuse our friendship now a little

bit further with an, an inequality which is true, but extremely sloppy. But I promise

it'll make our lives just easier in some future calculations. So rather than 4m+2,

'cause 2's sorta getting on my nerves. Let's just call this. Utmost six N.

Because m is at least one. [sound] Okay, you have to admit it's true, 6MO is at

least 4M plus two. It's very sloppy, these numbers are not anything closer to each

other for M large but, let's just go ahead and be sloppy in the interest of future

simplicity. Okay. Now I don't expect anyone to be impressed with this rather

crude upper bound, the number of lines of code that the merge subroutine needs to

finish, to execute. The key question you recall was how many lines of code does

merge sort require to correctly sort the input array, not just this subroutine. And

in fact, analyzing Merge Sort seems a lot more intimidating, because if it keeps

spawning off these recursive versions of itself. So the number of recursive calls,

the number of things we have to analyze, is blowing up exponentially as we think

about various levels of the recursion. Now, if there's one thing we have going

for us, it's that every time we make a recursive call. It's on a quite a bit

smaller input then what we started with, it's on an array only half the size of the

input array. So there's some kind of tension between on the one hand explosion

of sub problems, a proliferation of sub problems and the fact that successive

subproblems only have to solve smaller and smaller subproblems. And resolute

resolving these two forces is what's going to drive our analysis of Merge Short. So,

the good news is, is I'll be able to show you a complete analysis of exactly how

many lines of code Merge Sort takes. And I'll be able to give you, and, in fact, a

very precise upper bound. And so here's gonna be the claim that we're gonna prove

in the remainder of this lecture. So the claim is that Merge Short never needs than

more than six times N. Times the logarithm of N log base two if you're keeping track

plus an extra six N operations to correctly sort an input array of N

numbers, okay so lets discuss for a second is this good is this a win, knowing that

this is an upper bound of the number of lines of code the merger takes well yes it

is and it shows the benefits of the divide and conquer paradigm. Recall. In the

simpler sorting methods that we briefly discussed like insertion sort, selection

sort, and bubble sort, I claimed that their performance was governed by the

quadratic function of the input size. That is they need a constant times in the

squared number of operations to sort an input array of length N. Merge sort by

contrast needs at most a constant times N times log N, not N squared but N times

log N lines of code to correctly sort an input array. So to get a feel for what

kind of win this is let me just remind you for those of you who are rusty, or for

whatever reason have lived in fear of a logarithm, just exactly what the logarithm

is. Okay? So. The way to think about the logarithm is as follows. So you have the X

axis, where you have N, which is going from one up to infinity. And for

comparison let's think about just the identity function, okay? So, the function

which is just. F(n)=n. Okay, and let's contrast this with a logarithm. So

what is the logorithm? Well, for our purposes, we can just think of a logorithm

as follows, okay? So the log of n, log base 2 of n is, you type the number N

into your calculator, okay? Then you hit divide by two. And then you keep repeating

dividing by two and you count how many times you divide by two until you get a

number that drops below one okay. So if you plug in 32 you got to divide five

times by two to get down to one. Log base two of 32 is five. You put in 1024 you have to

divide by two, ten times till you get down to one. So log base two of 1024 is ten and

so on, okay. So the point is you already see this if a log of a 1000 roughly is

something like ten then the logarithm is much, much smaller than the input.

So graphically, what the logarithm is going to look like is it's going to look like. A

curve becomes very flat very quickly, as N grows large, okay? So F(n) being log

base 2 of n. And I encourage you to do this, perhaps a little bit more

precisely on the computer or a graphing calculator, at home. But log is

running much, much, much slower than the identity function. And as a result,

sorting algorithm which runs in time proportional to n times log n is much,

much faster, especially as n grows large, than a sorting algorithm with a

running time that's a constant times n squared.