wk1

# 0wk1-Overview

WELCOME: Welcome to the Algorithms Specialization! Here's an overview of the first week of material.

INTRODUCTION: The first set of lectures for this week is meant to give you the flavor of the course, and hopefully get you excited about it. We begin by discussing algorithms in general and why they're so important, and then use the problem of multiplying two integers to illustrate how algorithmic ingenuity can often improve over more straightforward or naive solutions. We discuss the Merge Sort algorithm in detail, for several reasons: it's a practical and famous algorithm that you should all know; it's a good warm-up to get you ready for more intricate algorithms; and it's the canonical introduction to the "divide and conquer" algorithm design paradigm. These lectures conclude by describing several guiding principles for how we'll analyze algorithms in this course.

ASYMPTOTIC ANALYSIS: The second set of lectures for this week is an introduction to big-oh notation and its relatives, which belongs in the vocabulary of every serious programmer and computer scientist. The goal is to identify a "sweet spot" of granularity for reasoning about algorithms --- we want to suppress second-order details like constant factors and lower-order terms, and focus on how the running time of an algorithm scales as the input size grows large.

PREREQUISITES: This course is not an introduction to programming, and it assumes that you have basic programming skills in a language such as Python, Java, or C. There are several outstanding free online courses that teach basic programming. We also use mathematical analysis as needed to understand how and why algorithms and data structures really work. If you need a refresher on the basics of proofs (induction, contradiction, etc.), I recommend the lecture notes "Mathematics for Computer Science" by Lehman and Leighton (see separate Resources pages).

DISCUSSION FORUMS: The discussion forums play a crucial role in massive online courses like this one. If you have trouble understanding a lecture or completing an assignment, you should turn to the forums for help. After you've mastered the lectures and assignments for a given week, I hope you'll contribute to the forums and help out your fellow students. While I won't have time to carefully monitor the discussion forums, I'll check in and answer questions whenever I find the time.

VIDEOS AND SLIDES: Videos can be streamed or downloaded and watched offline (recommended for commutes, etc.). We are also providing PDF lecture slides (typed versions of what's written in the lecture videos), as well as subtitle files (in English and in some cases other languages as well). And if you find yourself wishing that I spoke more quickly or more slowly, note that you can adjust the video speed to accommodate your preferred pace.

HOMEWORK #1: The first problem set consists of 5 multiple choice problems, mostly about Merge Sort and asymptotic notation. The first programming assignment asks you to implement one or more of the integer multiplication algorithms covered in lecture.

SUGGESTED READINGS FOR WEEK 1: Abbreviations in suggested readings refer to the following textbooks:

CLRS - Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms (3rd edition)

DPV - Dasgupta, Papadimitriou, and Vazirani, Algorithms

KT - Kleinberg and Tardos, Algorithm Design

SW - Sedgewick and Wayne, Algorithms (4th edition)

CLRS: Chapters 2 and 3

DPV: Sections 0.3, 2.1, 2.3

KT: Sections 2.1, 2.2, 2.4, and 5.1

SW: Sections 1.4 and 2.2

# 0wk1-menu

I. INTRODUCTION

Welcome and Week 1 Overview10 分

Overview, Resources, and Policies10 分

Lecture slides10 分

Why Study Algorithms?4 分

Integer Multiplication8 分

Karatsuba Multiplication12 分

About the Course17 分

Merge Sort: Motivation and Example8 分

Merge Sort: Pseudocode12 分

Merge Sort: Analysis9 分

Guiding Principles for Analysis of Algorithms15 分

II. ASYMPTOTIC ANALYSIS

The Gist14 分

Big-Oh Notation4 分

Basic Examples7 分

Big Omega and Theta7 分

Additional Examples [Review - Optional]7 分

Problem Set #1

购买课程以解锁此项目。

测验: Problem Set #15 个试题

Programming Assignment #1

购买课程以解锁此项目。

测验: Programming Assignment #11 个试题

# 1-1-Why Study Algorithms

0:00 Hi, my name's Tim Roughgarden. I'm a professor here at Stanford University. And I'd like to welcome you to this first course on the Design and Analysis of Algorithms. Now, I imagine many of you are already clear on your reasons for taking this course. But let me begin by justifying this course's existence. And giving you some reasons why you should be highly motivated to learn about algorithms. So what is an algorithm anyways? Basically it's a set of well defined rules, a recipe in effect for solving some computational problem. Maybe you have a bunch of numbers and you want to rearrange them so that they're in sorted order. Maybe you have a roadmap and an origin and a destination And you want to compute the shortest path from that origin to that destination. May be you face a number of different tasks that need to be completed by certain deadlines and you ant to know in what order you should accomplish the task. So that you complete them all by their respective deadlines. 1:01 So why study algorithms? Well first of all, understanding the basics of algorithms and the related field of data structures is essential for doing serious work in pretty much any branch of computer science. This is the reason why here at Stanford, this course is required for every single degree that the department offers. The bachelors degree the masters degree and also the PHD. To give you a few examples routing and communication networks piggybacks on classical shortest path algorithms. The effectiveness of public key cryptography relies on that of number-theoretic algorithms. Computer graphics needs the computational primitives supplied by geometric algorithms. Database indices rely on balanced search tree data structures. Computational biology uses dynamic programming algorithms to measure genome similarity. And the list goes on. 1:59 Second, algorithms play a key role in modern technological innovation. To give just one obvious example, search engines use a tapestry of algorithms to efficiently compute the relevance of various webpages to it's given search query. The most famous such algorithm is the page rank algorithm currently in use by Google. 2:21 Indeed in a December 2010 report to the United States White House, the President's counsel of advisers on science and technology argued that in many areas performance gains due to improvements in algorithms have vastly exceeded event he dramatic performance gains due to increased processor speeds. 2:44 Third, although this is outside of the score, the scope of this course. Algorithms are increasingly being used to provide a novel lens on processes outside of computer science and technology. For example, the study of quantum computation has provided a new Computational viewpoint on quantum mechanics. Price fluctuations in economic markets can be fruitfully viewed as an algorthmic process and even evolution can be usefully thought of as a surprisingly effect search algorthim. The last two reasons for studying algorthims might sound flippant but both have more than a grain of truth to them. I don't know about you, but back when I was a student, my favorite classes were always the challenging ones that, after I struggled through them, left me feeling a few IQ points smarter than when I started. I hope this course provides a similar experience for many of you. Finally, I hope that by the end of the course I'll have converted some of you to agree with me that the design and analysis of algorithms is simply fun. It's an endeavor that requires a rare blend of precision and creativity. It can certainly be frustrating at times, but it's also highly addictive. So let's descend from these lofty generalities and get much more concrete. And let's remember that we've all been learning about and using algorithims since we were little kids.

# 1-1-chs

0:00 嗨,我的名字叫Tim Roughgarden. 我是斯坦福大学的教授 欢迎参加 《算法的分析与设计（1）》课程 欢迎参加 《算法的分析与设计（1）》课程 我想大多数同学都很清楚你们学习本门课程的理由是什么 但还是让我解释一下这门课程为什么存在 让你知道你必须有足够大的动力来学习本门课程 让你知道你必须有足够大的动力来学习本门课程 到底什么是算法？ 从根本上来说它是一系列被精确定义的规则和方法 用来解决一些与计算机相关的问题 或许你有一串数字，你想去重新给这串数字排序 让它们按顺序排列 可能你有一张行车地图以及你的起点和终点 你想计算从七点到终点的的最短路径 也可能你面对若干任务 它们需要在截止日期之前完成 你想知道你应该按照什么样的顺序 完成这些任务 你应该完成你的任务按照它们各自的截止日期 1:01 所以，为什么学习算法？ 当你在做一些计算机领域的重要工作时 理解基础的算法和数据结构是很重要的 这就是为什么在斯坦福 每种学位的学生都要参加这门课 无论是本科生，研究生还是博士生 首先提几个关于算法的例子 routing和communication network 依赖于经典的最短路径算法 公钥加密的高效借助于数论算法 电子计算机制图需要几何算法提供的 运算初级指令 运算初级指令 数据库的索引依赖于平衡二叉树的数据结构 计算生物学使用动态规划测量基因组的相似性 这样的例子还有很多 1:59 第二，算法扮演了重要的角色 在现代的科技创新中 最显著的例子是 搜索引擎使用tapestry算法 高效的计算搜索产生的各种网页之间的相关性 最著名的算法就是 目前谷歌公司使用的page rank算法 2:21 确实在2010年的12月根据白宫的报告 总统顾问在科学和科技领域的建议声明 许多领域业绩的提升是因为 算法效率的提升 算法效率的提高甚至已经超出了 处理器提升的速度 2:44 第三，尽管这一点超出了本门课程的范围 算法正在越来越多的被运用到计算机科学与技术 以外的领域中 例如，对量子计算机的研究 它为量子力学的研究提供了一种新的计算视角 市场经济的价格波动可以被视为一种算法 甚至是evolution可以影响查找算法 最喜欢的总是那些最有挑战性的课程。 当我艰难的完成它们后 最后的两个学习算法的理由可能会听起来有些轻率 我不了解你，但当我还是个学生的时候 我最喜欢的课程总是那些具有挑战性的 但是当我费了很大劲学完后，你很有可能会觉得自己变得更聪明了！最后， 我希望这门课可以提供给你们和我一样的体验 最后我希望在课程结束的时候 最后，在课程结束的时候 我会让你们觉得《算法的分析与设计》是一门很有趣的课程 这门课需要严谨的态度以及创造性。 有时候你会感到有点沮丧 来讲一些更切实际的东西。 回想一下 但是他也很可能让你上瘾 让我们跳出这些一般性的总结，想想具体的情况 在差不多三年纪的时候，你学会了两个数的乘法 从我们很小的时候，我们就学过算法了

# 1-1

Hi, my name's Tim Roughgarden. I'm a professor here at Stanford University. And I'd like to welcome you to this first course on the Design and Analysis of Algorithms. Now, I imagine many of you are already clear on your reasons for taking this course. But let me begin by justifying this course's existence. And giving you some reasons why you should be highly motivated to learn about algorithms. So what is an algorithm anyways? Basically it's a set of well defined rules, a recipe in effect for solving some computational problem. Maybe you have a bunch of numbers and you want to rearrange them so that they're in sorted order. Maybe you have a roadmap and an origin and a destination And you want to compute the shortest path from that origin to that destination. May be you face a number of different tasks that need to be completed by certain deadlines and you ant to know in what order you should accomplish the task. So that you complete them all by their respective deadlines. So why study algorithms? Well first of all, understanding the basics of algorithms and the related field of data structures is essential for doing serious work in pretty much any branch of computer science. This is the reason why here at Stanford, this course is required for every single degree that the department offers. The bachelors degree the masters degree and also the PHD. To give you a few examples routing and communication networks piggybacks on classical shortest path algorithms. The effectiveness of public key cryptography relies on that of number-theoretic algorithms. Computer graphics needs the computational primitives supplied by geometric algorithms. Database indices rely on balanced search tree data structures. Computational biology uses dynamic programming algorithms to measure genome similarity. And the list goes on. Second, algorithms play a key role in modern technological innovation. To give just one obvious example, search engines use a tapestry of algorithms to efficiently compute the relevance of various webpages to it's given search query. The most famous such algorithm is the page rank algorithm currently in use by Google. Indeed in a December 2010 report to the United States White House, the President's counsel of advisers on science and technology argued that in many areas performance gains due to improvements in algorithms have vastly exceeded event he dramatic performance gains due to increased processor speeds. Third, although this is outside of the score, the scope of this course. Algorithms are increasingly being used to provide a novel lens on processes outside of computer science and technology. For example, the study of quantum computation has provided a new Computational viewpoint on quantum mechanics. Price fluctuations in economic markets can be fruitfully viewed as an algorthmic process and even evolution can be usefully thought of as a surprisingly effect search algorthim. The last two reasons for studying algorthims might sound flippant but both have more than a grain of truth to them. I don't know about you, but back when I was a student, my favorite classes were always the challenging ones that, after I struggled through them, left me feeling a few IQ points smarter than when I started. I hope this course provides a similar experience for many of you. Finally, I hope that by the end of the course I'll have converted some of you to agree with me that the design and analysis of algorithms is simply fun. It's an endeavor that requires a rare blend of precision and creativity. It can certainly be frustrating at times, but it's also highly addictive. So let's descend from these lofty generalities and get much more concrete. And let's remember that we've all been learning about and using algorithims since we were little kids.

# 1-2-Integer Multiplication

0:01 Sometime when you were a kid, maybe say third grade or so, you learned an Algorithm for multiplying two numbers. Maybe your third grade teacher didn't call it that, maybe that's not how you thought about it. But you learned a well defined set of rules for transforming input, namely two numbers into an output, namely their product. So, that is an algorithm for solving a computational problem. Let's pause and be precise about it. Many of the lectures in this course will follow a pattern. We'll define a computational problem. We'll say what the input is, and then we'll say what the desired output is. Then we will proceed to giving a solution, to giving an algorithm that transforms the input to the output. When the integer multiplication problem, the input is just two, n-digit numbers. So the length, n, of the two input integers x and y could be anything, but for motivation you might want to think of n as large, in the thousands or even more, perhaps we're implementing some kind of cryptographic application which has to manipulate very large numbers. 1:12 We also need to explain what is desired output in this simple problem it's simply the product x times y. So a quick digression so back in 3rd grade around the same I was learning the Integer Multiplication Algorithm. I got a C in penmanship and I don't think my handwriting has improved much since. Many people tell me by the end of the course. They think of it fondly as a sort of acquired taste, but if you're feeling impatient, please note there are typed versions of these slides. Which I encourage you to use as you go through the lectures, if you don't want to take the time deciphering the handwriting. Returning to the Integer Multiplication problem, having now specified the problem precisely, the input, the desired output. We'll move on to discussing an algorithm that solves it, namely, the same algorithm you learned in third grade. The way we will assess the performance of this algorithm is through the number of basic operations that it performs. And for the moment, let's think of a basic operation as simply adding two single-digit numbers together or multiplying two single digit numbers. We're going to then move on to counting the number of these basic operations performed by the third grade algorithm. As a function of the number n of digits in the input. 2:36 Here's the integer multiplication algorithm that you learned back in third grade Illustrated on a concrete example. Let's take say the numbers 1, 2, 3, 4 and 5, 6, 7, 8. As we go through this algorithm quickly, let me remind you that our focus should be on the number of basic operations this algorithm performs. As a function of the length of the input numbers. Which, in this particular example, is four digits long. So as you'll recall, we just compute one partial product for each digit of the second number. So we start by just multiplying 4 times the upper number 5, 6, 7, 8. So, you know, 4 times 8 is 32, 2 carry to 3, 4 times 7 is 28, with the 3 that's 31, write down the 1, carry the 3, and so on. When we do the next partial product, we do a shift effectively, we add a 0 at the end, and then we just do exactly the same thing. And so on for the final two partial products. [SOUND] And finally, we just add everything up. [SOUND], what you probably realized back in third grade, is that this algorithm is what we would call correct. That is, no matter what integers x and y you start with If you carry out this procedure, this algorithm. And all of your intermediate computations are done properly. Then the algorithm will eventually terminate with the product, x times y, of the two input numbers. You're never going to get a wrong answer. You're always going to get the actual product. Well, you probably didn't think about was the amount of time needed to carry out this algorithm out to its conclusion to termination. That is the number of basic operations, additions or multiplications of single digit numbers needed before finishing. So let's now quickly give an informal analyses of the number of operations required as a function of the input length n. 4:50 Let's begin with the first partial product, the top row. How did we compute this number 22,712? Well we multiplied 4 times each of the numbers 5, 6, 7 and 8. So that was for basic operations. One for each digit at the top number, plus we had to do these carries. So those were some extra additions. But in any case, this is at most twice times the number of digits in the first number. At most two end basic operations to form this first partial product. And if you think about it there's nothing special about the first partial product. The same argument says that we need at most 2 n operations to form each of the partial products of which there are again n, one for each digit of the second number. Well if we need at most two n operations to compute each partial product and we have n partial products. That's a total of at most two n squared operations to form all of these blue numbers, all of the partial products. Now we're not done at that point. We still have to add all of those up to get the final answer, in this case 7,006,652. And that's final addition requires a comparable number of operations. Roughly, another say two n squared, at most operations. So, the upshot, the high level point that I want you to focus on, is that as we think about the input numbers getting bigger and bigger. That is as a function of n the number of digits in the input numbers. The number of operations that the Grade-School Multiplication Algorithm performs, grows like some constant. Roughly 4 say times n squared. That is it's quadratic in the input length n. For example, if you double the size of the input, if you double the number of digits in each of the two integers that you're given. Then the number of operations you will have to perform using this algorithm has to go up by a factor of four. Similarly, if you quadruple the input length, the number of operations going, is going to go up by a factor of 16, and so on. 7:02 Now, depending on what type of third grader you were. You might well of accepted this procedure as the unique or at least the optimal way of multiplying two numbers together to form their product. Now if you want to be a serious algorithm designer. That kind of obedient tumidity is a quality you're going to have to grow out of. And early and extremely important textbook on the design and analysis of algorithms was by Aho, Hopcroft, and Ullman. It's about 40 years old now. And there's the following quote, which I absolutely adore. So after iterating through a number of the algorithm design paradigms covered in the textbook. They say the following, perhaps the most important principle of all, for the good algorithm designer is to refuse to be content. And I think this is a spot on comment. I might summarize it a little bit more succinctly. As, as an algorithm designer you should adopt as your Mantra the question, can we do better? This question is particularly apropos when your'e faced with a naive or straight-forward solution to a computation problem. Like for example, the third grade algorithm for integer multiplication. The question you perhaps did not ask yourself in third grade was, can we do better than the straight forward multiplication algorithm? And now is the time for an answer.

# 1-2-chs

0:01 在你还是个三年级小孩的时候 你学会了计算两个数字乘积的一种算法 也许你的小学老师不称它为算法 也许你没想过它是一种算法 但是你的确学到了一套定义精确的规则 它可以将输入数据 也就是两个数字 转化为输出 也就是它们的乘积 这就是一种解决计算问题的算法 这里我们停一停 详细说明一下 本课程中许多讲座会沿用一种模式 首先 我们明确要研究的计算问题 什么是输入 什么是期望的输出 然后我们会给出解决方案 即给出一种算法 使输入转化为输出 在整数乘法中 输入只是两个n位数x和y 数字x和y的位数n是任意的 但是如果想来点刺激的 你可以想象n很大 比如几千或者更多 像在进行某种加密时 就必须使用很大的数字 1:12 我们还需要解释什么是期望输出 在这个例子里很简单 就是x和y的乘积 插句题外话 我三年级的时候 在学习整数乘法的算法的同时 我书法课得了C 并且之后我也 不觉得我的字有多大长进 许多人在课程结束后告诉我 他们还倒挺喜欢我这奇葩手写体 但是如果你欣赏不了 网站上也有这些幻灯片的印刷体版本 如果觉得辨认手写体太麻烦 建议你在听讲座的过程中参阅它们 如果觉得辨认手写体太麻烦 建议你在听讲座的过程中参阅它们 回到整数乘法上 我们现在已经准确定义了问题 包括输入和期望的输出 现在就要讨论解决问题的算法了 也就是你小学三年级学的那玩意儿 我们评估该算法性能的标准 是它需要执行的基本运算的次数 这里的基本运算可以理解为将两个 个位数相加或相乘 然后我们要研究一下 输入长度为n时 若采用三年级算法 需要执行的基本运算次数 即它与n的函数关系 2:36 下面是一个具体的例子 就用1234和5678这两个数吧 提醒一下 在快速演示这个算法时 我们的关注点应该在于 这个算法做了多少次基本运算 基本运算的次数与输入长度 有怎样的函数关系 在这个例子里，输入长度是4 正如你记得的那样 我们用下面的数的每一位 分别与上面的数相乘 得到几个部分积 首先 我们把4与上面的5678相乘 四八三十二 3进到十位 四七二十八 加3为31 写下1 进3 以此类推 在计算下一个部分积时 有一点小变化 即在个位加一个零 其他的和上个计算相同 即在个位加一个零 其他的和上个计算相同 后面两个部分积的计算过程同上 最后 把四个部分积加在一起 你三年级的时候已经意识到 这种算法总是正确的 也就是说 不管整数x和y是多少 只要你执行这个算法 并保证中间的每一步计算都正确 那么这个算法最终会得到输入的 两个数据之积 即x乘y 那么这个算法最终会得到输入的 两个数据之积 即x乘y 你永远不会得到错误答案 它总是正确的 然而 你也许从未考虑过 从开始执行这个算法 到得到最终答案 需要多长时间 也就是要做多少基本运算 亦即需要做多少次单个数字的加法或乘法 那么 现在我们来快速地大致分析一下 基本运算的次数与输入长度 有怎样的函数关系 4:50 我们先从第一个部分积开始 我们计算22712这个数字的时候 是将4分别 与5 6 7 8这四个数字相乘 这就是4次基本运算了 下面的数的每一位都与上面的数相乘 乘积十位上的数字还要进上去 进位又涉及一些额外的加法 但是任何情况下 基本运算的数量最多是上面乘数长度的2倍 即最多运算2n次便可得出第一个部分积 第一个部分积并非特例 同理可得 得到每一个部分积都最多需要2n步基本运算 而部分积的总数为n 即与下面乘数的长度相等 也就是说 计算每个部分积需要2n次基本运算 且一共有n个部分积 所以要做2n^2次基本运算 才能得到图中所有蓝色的数字 也就是所有的部分积 然而这还不算完 我们还要把这些部分积加起来 才能得到最终结果 在这个例子中是7006652 这一步求和需要的运算量与之前一样多 也是最多2n^2次 所以一个很重要的结论是 如果采用三年级所学的乘法算法 那么随着输入的数据逐渐增大 基本运算数的增长方式是固定的 它与输入数据的长度具有函数关系 即基本操作数等于4n^2 是输入长度n的二次函数 例如 如果你把输入的两个整数的长度分别加倍 例如 如果你把输入的两个整数的长度分别加倍 那么采用这个算法的话 基本运算数会变为原来的4倍 类似地 如果把输入的两数长度 分别变为原来的4倍 那么基本运算数会变为原来的16倍 以此类推 7:02 现在 无论你是哪一种小学三年级学生 或许你都会觉得 计算两整数之积的方法只有这一种 或者至少最好用的是这一种 然而如果你想成为一个真正的算法设计师 这种顺从陈规的想法是一定要抛弃的 关于算法设计与分析 Aho, Hopcroft 和 Ullman 很早之前写过一本十分重要的书 该书已经出版了40年了 书中有一句话我非常喜欢 就是这句 在列举了一些的算法设计范例后 作者这样说道 "对于一个优秀的算法设计者而言 最重要的原则就是拒绝满足" 此言得之 更简洁地总结一下 作为一个算法设计者 你的座右铭应该是 我们可以做到更好吗? 面对一个幼稚的、直接的计算题解法时 这句话尤其适用 比如说 整数乘法的三年级算法 你三年级的时候 不会问自己 有没有比这个直接的算法更好的方法? 我们下节课揭晓答案 翻译：JennySquirrel | 审阅： Cousera Global Translator Community

# 1-3-Karatsuba Multiplication

0:02 If you want to multiply two integers, is there a better method than the one we learned back in third grade? To give you the final answer to this question, you'll have to wait until I provide you with a toolbox for analyzing Divide and Conquer algorithm a few lectures hence. What I want to do in this lecture is convince you that the algorithm design space is surprisingly rich. There are certainly other interesting methods of multiplying two integers beyond what we learned in third grade. And the highlight of this lecture will be something called Karatsuba multiplication. Let me introduce you to Karatsuba multiplication through a concrete example. I am going to take the same pair of integers we studied last lecture, 1, 2, 3, 4, 5, 6, 7, 8. I am going to execute a sequence of steps resulting in their products. But, that sequence of steps is going to look very different than the one we undertook during the grade school algorithm, yet we'll arrive at exactly the same answer. 1:04 The sequence of steps will strike you as very mysterious. It'll seem like I'm pulling a rabbit out of the hat, and the rest of this video will develop more systematically what exactly this Karatsuba multiplication method is, and why it works. But what I want you to appreciate already on this slide is that the algorithm design space is far richer than you might expect. There's this dazzling array of options for how to actually solve problems like integer multiplication. 1:34 Let me begin by introducing some notation for the first and second halves of the input numbers x and y. So the first half of x, that is 56- we're going to regard as a number in its own right called a. Similarly b will be 78, c will be 12, and d will be 34. 1:53 I'm going to do a sequence of operations involving only these double digit numbers a b c and d. And then after a few such operations I will collect all of the terms together in a magical way resulting in the product of x and y. First let me compute the product of a times c and also the product of b times d. I'm going to skip the elementary calculations, and just tell you the answer. So you can verify that a times c is 672, where as b times d is 2652. Next I'm going to do something even still more inscrutable. I'm going to take the sum of a and b. I'm going to take the sum of c and d. And then I'm going to compute the product of those two sums. That boils down to computing the product of 134 and 46. Mainly at 6164. Now, I'm going to subtract our first two products from the results of this computation. That is, I'm going to take 6164. Subtract 2652, and subtract 672. You should check that if you subtract the results of the first 2 steps from the result of the 3rd step, you get 2840. Now, I claim that I can take the results of step 1, 2 and 4 and combine them into super simple way to produce the product of X and Y. Here's how I do it. I start with the first product, ac. And I pad it with four zeros. I take the results of the second step, and I don't pad it with any zeros at all. And I take the result of the fourth step, and I pad it with two zeros. If we add up these three quantities, from right to left. We get two, five, six. Six, zero, zero, seven. If you go back to the previous lecture you'll note that this is exactly the same output as the great school algorithm, that this is in fact the product of one, two, the, three, four and five, six, seven, eight. So let me reiterate that you should not have any intuitions for the computations I just did, you should not understand what just went down on this slide. Rather I hope you feel some mixture of bafflement and intrigue but, more the point I hope you appreciate that the third grade algorithm is not the only game in town. There's fundamentally different algorithms for multiplying integers than what you learned as a kid. Once you realize that, once you realize how rich the space of algorithms is, you have to wonder Can we do better than that third grade algorithm? In fact, does this algorithm already do better that the third grade algorithm? Before I explain full-blown Karatsuba multiplication, let me begin by explaining a simpler, more straightforward recursive approach. To integer multiplication. Now, I am assuming you have a bit of programming background. In particular, that you know what recursive algorithms are. That is, algorithms which invoke themselves as a subroutine with a smaller input. So, how might you approach the integer multiplication problem recursively? Well the input are two digits. Each two numbers. Each has two digits. So to call the algorithm recursively you need to perform inputs that have smaller size, less digits. Well, we already were doing that in the computations on the previous slide. For example the number 5678 we treated the first half of digits as 56 as a number in its own right and similarly 78. 5:33 In general, given a number x with n digits. In can be expressed decomposed, in terms of two, n over two digit numbers. Namely as A, the first half of the digits shifted appropriately. That is multiplied by ten raised to the power, n over two. Plus the second half of the digits b. In our example, we had a equal to 56, 78 was b. N was 4, so 10 to the n over 2 was 100, and then c and d were 12 and 34. What I want to do next is illuminate the relevant recursive calls. To do that, let's look at the product, x times y. Express it in terms of these smaller numbers, a, b, c, and d, and do an elementary computation. 6:17 Multiplying the expanded versions of x and y, we get an expression with three terms. One shifted by n, 10 raised to the power n, and the coefficient there is a times c. We have a term that's shifted by 10 to the n over 2, and that has a coefficient of ad and also plus bc. And bringing up the rear, we have the term b times d. We're going to be referring to this expression a number of times, so let me both circle it and just give it a shorthand. We're going to call this expression star. 6:52 One detail I'm glossing over for simplicity, is that I've assumed that n is an even integer. Now, if n is an odd integer, you can apply this exact same recursive approach to integer multiplication. In the straightforward way, so if n was 9 then you would decompose one of these input numbers into say the first five digits and the later four digits and you would proceed in exactly the same way. Now the point of the expression star is if we look at it despite being the product of just elementary algebra, it suggests a recursive approach to multiplying two numbers. If we care about the product Of X and Y, why not, instead, compute this expression star, which involves only the products of smaller numbers, A, B, C and D. You'll notice, staring at the expression star, there are 4 relevant products, each involving a pair of these smaller numbers. Namely AC, AD, BC, and BD . So why not compute each of those four products recursively. After all, the inputs will be smaller. And then once our four recursive calls come back to us with the answer, we can formulate the rest of expression star in the obvious way. We just pad a times c with n zeros at the end. We add up a, d, and bc, using the grade school algorithm, and pad the result with n over two zeros, and then we just sum up these returns, again using the grade school addition, and algorithm. So the one detail missing, that I've glossed over, required to turn this idea into a bonafide recursive algorithm, would be to specify a base case. As I hope you all know, recursive algorithms need a base case. If the input is sufficiently small, then you just immediately compute the answer rather than recursing further. Of course, recursive algorithms need a base case so they don't keep calling themselves til the rest of time. So for integer multiplication, which the base case, well, if you're given two numbers that have the just one digit each. Then you just multiply them in one basic operation and return the result. So, what I hope is clear at the moment is that there is indeed a recursive approach to solving the integer multiplication algorithm resulting in an algorithm which looks quite different than the one you learned in third grade, but which nevertheless you could code up quite easily in your favorite programming language. Now, what you shouldn't have any intuition about is whether or not this is a good idea or a completely crackpot idea. Is this algorithm faster or slower than the grade school algorithm? You'll just have to wait to find out the answer to that question. Let's now refine this recursive algorithm, resulting in the full-blown Karatsuba multiplication algorithm. To explain the optimization behind Karatsuba multiplication, let's recall the expression we were calling star on the previous slide. So, this just expressed the product of x and y in terms of the smaller numbers a, b, c, and d. In this straight forward recursive algorithm we made four recursive calls to compute the four products which seemed necessary to value, to compute the expression star. But if you think about it, there's really only three quantities in star that we care about, the three relevant coefficients. We care about the numbers ad and bc. Not per se, but only in as much as we care about their sum, AD plus BC. So this motivates the question, if there's only 3 quantities that we care about, can we get away with only 3 rather than 4 recursive calls. It turns out that we can and here's how we do it. 10:31 The first coefficient a c and the third coefficient b d, we compute exactly as before, recursively. Next, rather than recursively computing a d or b c, we're going to recursively compute the product of a plus b and c plus d. If we expand this out, this is the same thing as computing ac plus ad plus bc plus bd. Now, here is the key observation in Karatsuba Multiplication, and it's really a trick that goes back to the early 19th Century mathematician, Gauss. Let's look at the quantity we computed in step 3 and subtract from it. The two quantities that we already computed in steps one and two. 11:17 Subtracting out the result of step one cancels the a c term. Subtracting out the result of step two, cancels out the bd term, leaving us with exactly what we wanted all along, the middle coefficient a d plus b c. And now in the same that on the previous slide we have a straightforward recursive algorithm making four recursive calls, and then combining them in the obvious way. Here we have a straightforward recursive algorithm that makes only three recursive calls. And on top of the recursive calls does just great school addition and subtraction. So you do this particular difference between the three recursively computed products and then you do the shifts, the padding by zeros, and the final sum as before. 12:04 So that's pretty cool, and this kind of showcases the ingenuity which bears fruit even in the simplest imageable computational problems. Now you should still be asking the question Yeah is crazy algorthim really faster than the grade school algorithm we learn in 3rd grade? Totally not obvious, we will answer that question a few lecture hense and we'll answer it in a special case of an entire toolbox I'll provide you with to analyze the running time of so called divide and conquer algorithms like Karatsuba multiplication, so stay tuned.

# 1-3-chs

0:02 如果你想计算两个整数之积 除了三年级学的那种方法外 还有没有其他更好的方法？ 为了得出最终结论 我会先提供给你一些工具 为之后提到的分治算法做铺垫 本节课的目的是使你树立一个信念 就是算法设计的空间十分广阔 除了三年级学的方法之外 还有好多计算两整数之积的方法 本节课的核心是介绍一种乘法 叫做Karatsuba乘法 下面就让我通过具体的例子 介绍一下Karatsuba乘法 我们接着用上节课的那两个数:1234和5678 我将通过执行一系列步骤 得到它们的乘积 但是 这些步骤明显不同于 我们小学所学的乘法运算步骤 然而得到的答案将完全相同 1:04 这一系列步骤会使你感到十分迷惑 就像从帽子里变出一只兔子一样 而本节课其余的时间 将更系统地讲解Karatsuba算法究竟是什么 以及它为什么有效 但是首先你要明确一点 那就是 算法设计的空间 比你想象的广阔的多 计算整数乘法之类的问题 有非常非常多的方法 1:34 首先 我要把x和y的前两位和后两位 分别用字母来表示 例如x的前两位 也就是56 56就用a来表示 类似地 b代表78 c代表12 d代表34 1:53 接下来我要做一系列运算 参与运算的只有a b c和d 之后我会将一系列运算的结果 以某种神奇的方式汇总 最终得到x和y的乘积 首先 我要计算ac的乘积 然后计算bd的乘积 我将跳过这些简单的计算 直接告诉你 a乘c等于672 b乘d等于2652 接下来第三步中 我要做一件更难以理解的事 我要分别算出ab之和 cd之和 然后计算上述两个和之积 也就是计算134与46的积 结果是6164 最后 我要从这个结果中减去ac之积 再减去bd之积 也就是用6164 减去2652 再减去672 如果你的确是用第3步减去了前两步的结果 你会得到2840 现在 利用第1,2,4步的结果 可以通过一种非常简单的方法得到x和y的乘积 做法是这样的 我们从第一步的结果 即ac的乘积开始 在它后面加四个零 然后我们取第二步的结果 不做任何改动 接着取第四步的结果 在后面加两个零 最后把这三个数加起来 从右到左 得到2,5,6,6,0,0,7 如果你回顾之前的课 你会发现 这个结果跟你用小学算法 算出的结果完全相同 事实上 这个数字就是1234与5678的乘积 这里我重复一遍 你可能无法 凭直觉理解这些计算过程 你可能不清楚刚才发生了什么 然而除了觉得它像一个谜团或者阴谋之外 我更希望你体会到 三年级的算法不是唯一的解决方案 还有很多计算整数乘积的方法 它们和你小时候学的那种均有天壤之别 一旦你意识到算法设计的空间如此广阔 你必然会思考 能不能找到 比三年级的算法更好的方法? 或者事实上 是否上述算法 已经优于三年级算法了? 在解释Karatsuba乘法的原理之前 让我们从一个更简单、更直接的递归方法开始 这里我假设你有一点编程基础 具体讲就是 你知道什么是递归算法 就是一种在子过程中不断调用自身 同时缩小输入的算法 那么 如何将递归法应用于乘法问题中呢? 你看 输入的数据是两个四位数 你看 输入的数据是两个四位数 为了将递归法应用于算法中 你需要将输入数字变小 我们之前已经做过这件事情了 比如说 我们把5678拆成了两半 也就是56和78 5:33 一般而言 一个长度为n的数x 可以被拆分成两个长度为n/2的数 x的前一半等于a\*10^(n/2) x的前一半等于a\*10^(n/2) 后一半等于b 本例中a=56 b=78 n=4 10^(n/2)=100 c=12 d=34 接下来我将解释有关的递归调用 我们要把x和y的乘积 用a b c d表示 并做一个简单的计算 6:17 将x和y的展开式相乘 我们得到一个由三部分组成的多项式 第一项含10^n 系数是ac 第二项含10^n/2 系数是ad+bc 最后一项为bd 我们将多次提到这个表达式 所以我要把它圈起来 还要给它一个简称 就叫它星式吧 6:52 为了简化问题 我掩盖了一件事情 就是我假设了n是偶数 不过如果n是奇数的话 你一样可以用这个递归的方法计算整数乘法 直接点说 就是如果n=9 你可以将其中一个输入数据的前5位和后4位拆开 其他步骤与上述完全相同 星式的重点在于 尽管它是基本代数运算的产物 但它隐含着一种递归的方法 如果你想得到x和y的乘积 为什么不先从星式入手呢 它只含有a b c d的乘积 盯着星式看一会 你会发现其中有四个相关的乘积 每个乘积都由两个更小的数字组成 也就是ac ad bc和bd 那么 为什么不先递归地计算出这四个乘积呢 毕竟这样一来 输入数据会小很多 在这四个递归调用返回给我们结果后 很明显 我们就可以计算星式的值了 我们在ac乘积后面加n个零 然后把ad和bc用小学学的方法相加 在结果后面加n/2个零 最后仍然用小学的方法 把这三个数加起来 我又掩盖了一个细节 就是真正的递归算法必须有一个基准情形 我希望你们都清楚 递归算法需要一个基准情形 如果输入的数据足够小 那么与其继续递归 不如直接计算答案 当然 递归算法本身也需要一个基准情形 来防止其无休止地调用自身 对于整数乘法 基准情形就比如 计算两个个位数的乘积 这时你直接采用基本运算 得到结果即可 所以 我现在想说明的是 的确存在一种递归法可以计算整数乘积 这种算法看起来和你三年级学的非常不同 然而用你最喜欢的编程语言 你可以轻松地将它编写出来 现在 对于这个算法是好是坏 你应该毫无概念 这个算法与三年级算法相比是快是慢? 等一下你就会知道答案 我们先来改进一下这个递归算法 从而得到成熟的Karatsuba乘法算法 为了解释对它的优化 我们来回忆一下之前提到的星式 星式用更小的数a b c和d 表示了x和y的乘积 在这个直接的递归算法中 我们执行了四次递归调用 来计算四个乘积 它们是计算星式所必须的 但是如果你仔细想想 星式中我们关心的只有三个量 即三个系数 我们关心ad和bc 本质上是在关心它们的和 即ad+bc 由此产生一个问题 既然我们关心的只有3个量 能不能只执行3次递归调用 而非4次? 事实上是可以的 下面是具体做法 10:31 第一个系数ac 和第三个系数bd 仍然和之前一样递归地计算 接下来 我们将递归地计算(a+b)(c+d) 而不是计算ad和bc 将(a+b)(c+d)展开 会得到 ac+bd+ad+bc 下面是Karatsuba乘法的关键 它其实是19世纪数学家高斯发明的一个技巧 我们用第3步的结果 减去前两步的结果 11:17 减去第一步的结果后 ac项不见了 减去第二步的结果后 bd项不见了 于是只剩下了ad+bc 即星式里中间项的系数 之前所讲的简单的递归算法中 我们执行了四次递归调用 并将四个结果组合在一起 现在我们又得到了一个简单的递归算法 它只需执行三次递归调用 且在此基础上只需做小学的加减法 所以你只需要对这三个递归结果 做一点特殊处理 然后做一下转换 在后面加零 最后和之前一样求和 12:04 这简直是在搞大新闻 它向我们展示了 即使在最简单的计算问题中 创造力也可以发挥作用 你也许还在问那个问题 这个奇葩的算法真的比三年级学的那种算得快吗? 这个实在不容易看出来 因此我们过几节课再来讲 到时候我会提供给你一整套工具 来分析所谓分治算法的运行时间 而这个问题会作为它的特例 敬请期待 翻译：JennySquirrel | 审阅： Cousera Global Translator Community

# 1-4-About the Course

0:00 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. Let's start with the specific topics that this course is going to cover. The course material corresponds to the first half of the ten week Stanford course. It's taken by all computer science undergraduates, as well as many of our graduate students. There will be five high level topics, and at times these will overlap. 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. The goal is to provide an introduction to and basic literacy in each of these topics. Much, much more could be said about each of them, than we'll have time for here. The first topic is the shortest, and probably also the driest. But it's a prerequisite for thinking seriously about the design and analysis of algorithms. 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. It turns out that the sweet spot for clear high level thinking about algorithm design, is to ignore constant factors and lower-order terms. And to concentrate on how well algorithm performance scales with large input sizes. Big O notation is the way to mathematize this sweet spot. Now, there's no one silver bullet in algorithm design. No single problem solving method that's guaranteed to unlock all of the computational problems that you're likely to face. That said, there are a few general algorithm design techniques. High level approaches to algorithm design that find successful application across a range of different domains. These relatively widely applicable techniques are the backbone of a general algorithms course like this one. In this course, we'll only have time to deeply explore one such algorithm design paradigm, namely that of the divide and conquer algorithms. In the sequel course as we'll discuss, there's two other major algorithms on paradigms to get covered. 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. So for example, in the last video. We saw two algorithms of this sort, two divide and conquer algorithms from multiplying two large integers. In later videos we will see a number of different applications. 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. In addition, we'll cover some powerful methods for reasoning about the running time of recursive algorithms like these. 2:37 As for the third topic. 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. 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. The canonical example is randomized quick sort, and that algorithm and analysis we will cover in detail in a few lectures. 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. 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. 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. 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. 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. 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. 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. 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. In this class we'll focus on primitives for competing connectivity information and also shortest paths. We'll also touch on how some primitives have been used to investigate the structure of information in social networks. Finally, data structures are often a crucial ingredient in the design of fast algorithms. A data structure's responsible for organizing data in a way that supports fast queries. Different data structures support different types of queries. 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. 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. There's two extremely useful data structures that we'll discuss in detail. The first is balanced binary search trees. 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. 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. 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. >> 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. The first part of this sequel course focuses on two more algorithm design paradigms. First of all, the design analysis of greedy algorithms with applications to minimum spanning trees, scheduling, and information theoretic coding. 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. The second part of the sequel course concerns NP complete problems, and what to do about them. 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. We'll discuss the theory of NP completeness, and, with a focus on what it means for you as an algorithm designer. 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. Of course there are plenty of important topics that can't be fit into either of these two five-week courses. Depending on the demand, there might well be further courses on more advanced topics. Following this course is going to involve a fair amount of time and effort on your part. So it's only reasonable to ask: What can you hope to get out of it? What skills will you learn? Well. Primarily, you know, even though this isn't a programming class per se, it should make you a better programmer. 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. 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. Second, while this is not a math class per se, we'll wind up doing a fair amount of mathematical analysis. And this in turn will sharpen your mathematical analytical skills. You might ask, why is mathematics relevant for a class in the design and analysis of algorithms, seemingly more of a programming class. 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. 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. Why we analyze the algorithms in the way that we do, why various super fast algorithms are in fact super fast, and so on. And it turns out that good algorithmic ideas usually require nontrivial mathematical analysis to understand properly. You'll acquire fundamental insights into the specific algorithms and data structures that we discuss in the course. And hopefully, many of these insights will prove useful, more generally, in your other work. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. How that expression naturally translated into a recursive algorithm. In particular, I was certainly assuming that you had some familiarity with recursive programs. 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. 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. [sound]. 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. 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. 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. 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. 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. 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. 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. 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. I also need you to be familiar with basic mathematical notation, like the standard quantifier and summation symbols. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. >> 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. 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. 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. 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. 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. 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. [sound]. 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. 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. 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. 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. 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.

# 1-4-chs

0:00 在这个视频里面 我将讨论这个课程的各个方面和主题 包括你将会获得的各种技能 需要的背景知识 课程相关材料以及可用的自我评估工具 我们以这个课程所涵盖的主题开始吧 该课程的内容与斯坦福大学校内课程的前五个星期一致 所有计算机科学专业的学生和很多毕业生学习这门课 该课程有5个主题 并且它们会有重叠部分 这5个主题是 分析算法性能的所需知识 分治算法设计模板 随机化算法设计 图论分析的基础知识 以及基本数据结构的运用与实现 课程的目标是 对上面的每个主题提供介绍及基本素养 详细讲解每个主题所需时间都比我们在这有的时间长得多 第一个主题最简短 也有可能最无趣 但这对于严肃地思考 算法的设计与分析是一个前提 这里的关键概念是大O符号 从概念上讲 它对于我们衡量一个算法运行时间的表现来说 是一个模型化的选择 它忽视了常数和低阶项 为更高阶段的算法设计提供了很大方便 而且大O符号关注大输入规模 将这种便利的分析数学化 而且不会有唯一的最优算法设计 没有任何一个解决问题的方式能够保证解决所有的 我们可能面对的算法问题 也就是说 几乎没有 所谓的万能算法设计技巧 算法设计的抽象思想 成功在不同领域广泛应用 这些相对宽泛的运用技巧是支柱 支撑着像这门课一样的算法课程 在这里 我们只有时间去深入的探索一个这样的 算法设计的范例 名叫分治法 以后的课程我们将谈论到另外的两个主要的算法范例 现在我们说说分治法算法 它是首先把一个问题 分成小的问题以待递归解决 然后快速的结合这些递归运算结果得到一个 你真正想要解决的问题的答案 例如 上一个视频中 我们看了两个算法 两个关于两个大整数相乘的分治算法 后面的视频中 我们将会看到很多不同的分治法应用 我们将知道如何设计快速的分治算法 用来解决 诸如矩阵乘法 最近点对和计算几何之类的问题 此外 我们还会覆盖一些有用的方法 用于分析像这些的递归算法的运行时间 2:37 第三个部分 随机算法指的就是从某种程度上来讲 在运行中投掷硬币的算法 如果你多次的运行它 计算同一个输入 它的运行过程将是不一样的 事实证明 这绝对不直观易懂 让一个算法在它内部进行随机 往往能得到简单 优雅 并且实用的算法 能够解决多种 算法问题 经典案例就是随机化快速排序 这个例子的算法和分析 我们将会用几节课来详细的讲解 随机化质数检测是我们会接触到的另一个杀手级应用 我们还会讨论图形分解的一种随机方法 最后我们会讲如何在哈希函数与哈希表分析中应用随机化 这门课程的主题之一 同时也是我希望你们能在课程中 学到的一项实际技能 就是了解一些操作数据用的计算方法 它们快到可以在某种程度上说几乎不花时间 这就是说 你在某一个算法上投入的时间 几乎不比你在检视或者读取数据上花的时间要多 当你有一个非常快速的算法时 它的运行时间与读取数据的速度几乎相同时 你应该准备好将它付诸应用 举例来说 在一个处理环节中 只要这个算法可能有用 你就应该把它准备好 随时调用 排序是一个典型的非常快速 接近不需耗时的算法 但是也存在着在处理更复杂的数据的优良算法 回忆一下 图是一种由若干顶点和 链接顶点的边组成的数据结构 图模拟不同种类的网络 所以即使图 比简单的数组复杂的多 还是有一些非常快的 分析图结构的方法 在这门课程中 我们会着重介绍 计算连通性信息和寻找最短路的方法 也会接触到一些方法 用来探索社交网络中的信息结构 最后 数据结构通常是优秀算法中重要的组成部分 一个数据结构应该有效地组织数据 以便快速对这些数据 进行处理 不同的数据结构支持不同的操作方式 我们假定你熟悉基础编程课程中会遇到的数据结构 包括数组 不定长数组 列表 栈和队列 最好是你已经稍微接触过了树和堆 或者你愿意在课程外 稍微阅读一下关于它们的材料 不过随着课程进行 我们也会 对上面提到的每种数据结构进行简短复习 我会细致讲解 两种极为重要的数据结构 第一种是平衡二叉搜索树 这些数据结构动态地保持一组元素的顺序 同时 支持很多种操作 这些操作的时间复杂度可以是对数级别的 第二个我们会讨论到的重要数据结构是哈希列表或 称作哈希图 这种结构记录一个动态的集合 支持极为快速的 插入和搜索 我们会讲这种数据结构一些权威经典的应用 以及在这种数据结构的典型实现运行的详细过程 在算法的设计分析中 有一些重要概念我们没时间 在这五周的课程中涉及到 这其中有一部分会 在后续课程：算法设计与分析2 中涉及到，这个课程和 斯坦福大学校内相关课程的后半部分吻合 这个后续课程的前半部分讨论了另外两种算法设计思想 首先就是贪心算法的设计 它用于最小生成树 时间调度问题和信息编码理论中 第二就是动态规划算法 例子包括 它在基因序列和社交网络最短路拟定中的应用 应用 后续课程的第二部分讨论NP完全问题 以及如何解决他们 NP完全问题是什么呢? 有一个你可能听过的著名数学猜想 叫做 "P不等于NP"猜想 NP完全问题是那种在这个猜想下 不能被高效算法解决的问题 我们会讨论 NP完全的理论 着重介绍它对你这样 一个算法设计者的意义 我们会讨论NP完全问题的解法 包括一些能够解决特殊情况的快速算法 高效的有可证效率保证的回溯算法 还有指数时间复杂度的 在本质上比暴力搜索优秀的算法 当然 还有一些 重要的主题 不能在这两个五周的课程中出现 根据需要 也许会有更多的课程 来介绍更高级的主题 你上这门课会花一定时间 和精力 所以 很重要的是问一句 你希望从这门课程中得到什么? 你会学到什么技能? 即使它本身不是编程课 它也会让你成为一个更好的程序员 你在描述和分析算法的时候会得到很多的练习 你会学到 算法设计思想 以及真正的和很多不同领域不同问题相关的 高水平的问题解决策略 还有 预测这种算法效率的工具 你会学到几个极其高效的 高速的处理数据的子程序 以及一些有用的 可以在你的程序中直接利用来组织数据的数据结构 第二 虽然这本身不是一节数学课 但我们会用到相当多的 数学分析 这会增强你的数学分析能力 你可能会问 为什么数学会与一门算法设计和分析课相关 它更像一门编程课 我来明确一下 我对于仅仅告诉你们知识点或者机械重复代码毫无兴趣 那些你们已经可以在网页或者好的编程书里面找到 我在这门课里的目的 以及我认为我能够最好地补充 那些你们可能已经得到的资源的方式 就是解释 为什么事情是这样的 为什么我们用这种方法分析算法 为什么各种超级快的算法实际上那么快等等 并且事实证明 良好的算法思路经常要求复杂的数学分析 才能充分地理解 你将获得对我们这门课讨论的 具体算法和数据结构的基本洞察力 并且希望 普遍而言 在你的其他工作上 大部分见解将能证明是有用的 第三 可能与你们中那些非计算机专业的人最相关 这门课会帮助你学习如何 "（按照）算法般思考" 实际上 学习算法之后很难不看到他们无处不在 无论你在乘电梯 观察一群小鸟 按照你的投资机会买卖股票 甚至是观察一个婴儿学习 正如我在上一个视频里所说 思考算法变得更有用 更流行 如果你不是CS之外的专业 比如生物 统计以及经济 第四 如果你有兴趣体验当一名计算机科学家 那么 某种程度上 你将一定想要知道 我们将要包含的所有主题的基本素养 实际上 让算法有趣的事情之一就是它真的让你觉得你正在学习 许多过去50年来最伟大的灵感 因此 学习完这门课之后 当有人抛出一个关于Dijkstra(狄杰斯特拉)算法的笑话时 你将不再感到是计算机科学的外行 你就会明白他们的意思 最后 毫无疑问学习这些材料有助于（回答）技术面试问题 清楚地讲 我在这里的唯一目的是教授你们算法 不是帮助你们准备面试 但是多年来 我的无数学生给我讲 掌握这门课中的概念如何帮他在被问到技术问题表现完美 这让我很高兴 我告诉你们 这是最基本的东西 那么 我对你们的期待是什么呢 诚实地讲 其实没什么期待 毕竟这不是一门免费在线课程的重点 就像这门课 任何人都可以学习并按照他们的意愿付出努力 作为老师 心里有一个或几个优等生是有用的 我想我继续讲 让你们明白我是怎样思考这些课程内容的 我认为我正在教授的学生是怎么样的 所以再说一遍 请不要感到沮丧如果你不符合优等生模板 我很高兴有这个机会教授你们关于算法的知识 不论你是什么样的学生 所以首先 我觉得至少你懂一些编程 例如 考虑前面的课 我们谈论了关于 一个递归方法求解两大数相乘 我提到了特定的数学表达式 当时我们标了星号并且用绿色圈了一下 那个表达式是如何自然地翻译成递归算法 尤其是 我当时已经假设你们熟悉一点递归程序 如果你接受我在那节课的陈述 如果你觉得 你可以写出整数递归相乘的代码 基于我给你们的高级大纲 那么你应该比较适合这门课 你应该做好准备了 如果你不适应我当时讲的内容 那么 你可能不适应这门课里我们谈论程序用的相对高级的概念 但是我鼓励 无论如何 继续看下面几个视频 尝试看看 能否尽力理解这些信息 现在 虽然我的内容针对那些懂编程的人 但是我没假设你要懂得任何一种特定的编程语言 任何一种标准命令式语言 比如C Java或者Python都可以 为了让尽量多的人可以跟上这些课 坦诚而言 也是为了促进 坦诚而言 也是为了促进 在相对抽象的概念层次思考编程的能力 我不会 我不会用任何一种特定编程语言描述算法 我只会用高级别的伪代码 或者简单点只用英语 我会假设你自己有能力把这样一种高阶的描述 用你最喜欢的语言表现出来 事实上 我非常鼓励每个人能够在课后把所有讨论过的算法 都用自己的语言去实现一遍 这会加强你的理解 感受这些算法的精妙之处 事实上 很多专业计算机科学家 和程序员都是在他们真正用程序实现了一个算法之后才感觉理解了它 我们在课程的任务中也会特别安排这种编程作业 换句话说 如果你只是在找一门能让你 把代码复制粘贴到你的程序中去运行 而完全不用去理解为什么这段程序能成功的课程 这门课程并不适合你 外面已经有不少 这类的书籍了 第二 我还希望你们至少有 一定程度的数学知识 即使 只是积累的一些知识碎片 更准确的说 我期望你能够 进行基本的逻辑论证 包括 推断证明和反证两种方式 你们还需要熟悉一些基本的数学记号 比如 一些标准量词 求和符号 比如 如果你了解一些离散概率分布 那么 随机算法相关的课程还有哈希对你来说会容易一些 但是 除去这样一些基本知识 这门课程会是独立的 你甚至不需要知道积分的计算 除了分析随机快速排序算法时突然单独出现一个简单积分 你们绝大部分人过去都学过数学 但是你可以 把这作为一种复习进修 进一步打磨 网络上有很多相关资料 并且我很希望你们能去探索然后发现自己喜欢的 不过 这里我可以先推荐一个很棒并且免费的课程 Mathematics for Computer Science(计算机科学中的数学) 作者是Eric Lehman和Tom Layden 这门课很容易搜到 课程里面涵盖了所有我们这门课需要了解知识 还有大量的 其它知识 为了我们这门课能够更广泛更容易的上手 我们会把课程依赖的其它材料尽可能的减到最少 这门课是相对独立的 并且我们会提供讲义的PPT和PDF 有时 我们也会提供一些额外的阅读材料 这门课没有教科书 不过 这并不是说不用看书 而是 大部分的材料包含在了一系列非常棒的算法书籍中 我在这里推荐四本 前三本对我的思维方式还有 授课方式都有很大的影响 因此是公认的经典书籍 关于第二本还有个很酷的事情 就是作者Dasgupta Papadimitriou和Vazirani 在网上发布了一个在线的免费版本 因此 如果你在网上搜一下书名和作者 你应该很容易搜到 同时 这也基本上是我推荐第四本书的原因 第四本书的作者也在网上发布了一个可用的 完整的版本 并且和我们将要讨论的材料很匹配 如果你想了解这门课程中知识点的细节 或者你想从另一个角度去思考我讲到的问题 这些书籍会是很好的资源 当然还有很多很棒的书籍 没有在这里一一列出 因为我希望你们 能够主动去探索发现适合你的 在课程中 我有时会要求你 去编程实现一个算法 然后解决一个具体的不能手算的问题 我们并不关心你用什么语言 用什么环境 我们只需要你能给出一个最终答案 也就是说 我不会做任何具体要求 你能处理这个问题即可 如果你需要一些编程环境的帮助和建议 我们建议你通过课程论坛 去向其它同学求助 最后我们再说一些关于课程评分的事情 这门课程本身并没有官方要求的评价标准 但我们每周还是会布置任务 有三个原因 第一是能够让你们自我检测一下 可以帮助你们了解自己对学习材料理解了多少 哪个知识点已经掌握 哪个还没有 第二个是想稍微施加一点压力 设置截止日期也是这个目的 以便让你们更有动力去完成所有的课程 设置截止日期还有一个很重要的原因 就是能够让 班级上的同学学习进度同步起来 这样当同学们 在班级论坛中讨论问题寻求帮助时也会更有效率 最后的一个原因就是满足那些想要挑战自我的同学 在学习了课程材料的同时 能够加深对算法的理解 这门课已经有成千上万个学生 因此能够自动评分是非常重要的 但是 现在像这类的在线免费课程才处于1.0时代 因此可以用的自动评分工具还相当稚嫩 所以 我们只能尽量做好 不过说老实话 这很难 甚至 通过现有的工具 几乎不可能从深层面去理解你算法的设计 因此 这门在线课程并不能够取代最原始的斯坦福校内版本 这上面给出的任务和考试要求 跟斯坦福教授的课程也会有所不同 为了弥补这个 我们会时常发布一些附加的算法设计问题 可能通过视频 也可能是补充的任务的形式 我们没办法对这些问题去评分 但是希望你有兴趣挑战它们 并且通过班级讨论的论坛跟同学们讨论 可能的解法 并最终尽可能的解决你关于课程 的所有疑问 让我们最后再明确一下我们的目的 就是尽可能的理解算法背后的意义

# 1-5-Merge Sort Motivation and Example

0:00 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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. 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". 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. 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. 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. 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. 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. 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. Okay? So we'll get a win, first sorting in Divide-and-Conquer, and Merge Sort is the algorithm that realizes that benefit. 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. 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. 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. 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. 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. 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, ... 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. 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. 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. 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. 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. 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. 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. 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? 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. 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? 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. 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. 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. 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. 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. 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.

# 1-5-chs

0:00 在这段视频中，我们要来讨论下如何分析一个算法。 我要以一个非常有名的排序算法来作为开场，这就是“归并排序“ (Merge Sort) 算法。 接着我会精确地告诉你归并排序最多需要多少操作数 来正确地把输入数列排序。 我觉得我得先对各位抱歉： 现在是2012年了，归并排序这个古老的算法看起来已经相当遥远了。 即使回到冯纽曼那个时代——1945年，归并排序也是众所周知的了。 所以，我为什么要在这个现代算法课程中用这么一个古老的算法来作为范例呢？ 这有很多原因： 第一，虽然我们之后会讨论到许多算法，归并排序是很古老，但是它也很好。 归并排序大概已经有六、七十年的历史了，但是我们一直用它来排序，它的确是个很实用的算法， 它也总是被列入很多标准库中，作为标准的排序算法。这就是第一个原因。 当然还有很多其他的原因，我会在后面仔细说明。 首先，在这个网上课程中，我们将看到很多应用于各个领域的通用型算法。 我们先来着重讨论“分治法” (Divide-and-Conquer) 算法设计。 分治法的基本原理是将原本的问题分成几个较小的问题，然后以递归的方法来解决他们， 接着，把这些结果结合起来以解决原本的问题。 归并排序直到今天都也许是最容易理解的分治法算法的一个例子。 它清晰地呈现出这类算法的设计理念、分析中得挑战以及其优点。 说到优点，你们可能已经知道排序是什么意思， 或许也知道一些排序算法，其中可能就有归并排序。 相较于其他简单的或者说是很直觉的排序算法，归并算法更有优势。 例如，你可能已经了解的三种排序算法——选择排序、插入排序、冒泡排序，虽然在这个课程中我们不会讲到它们 如果你还不了解这些排序算法，我希望你能从书上或者网上查一查。 我们来看看这三种看起来比较简单的排序算法。第一个叫做“选择排序”。 你将在输入数列中寻找最小的元素，然后重复这个过程， 所以选择排序的基本原理就是以线性方式来寻找最小值的过程。 接下来是“插入排序”，它虽然没有归并排序那么好，但是在有些情况下来说也是很实用的，我们会在之后谈到。 插入排序的基本原理是将已经排序的元素放在数列前面，并将下一个元素放在对应位置，重复这个过程直至全部排序。 像这样进行十个回合之后，排在最前面的十个元素都是已经排序了的。 当所有的元素都被处理过后，那么整个数列也就排好序了。 最后我们看到的是你也许已经了解的“冒泡排序”了，其基本原理是判断相邻的元素是否正确排列， 如果没有正确排列，就交换这对相邻的元素的位置，重复操作直到整个数列都排好序为止。 再次提醒，这些算法都是比归并算法更简单的排序算法， 但是它们的效率都不怎么高。如果有N个元素需要排序，它们的时间复杂度大约是N^2这个级别。 换句话来说，它们将花费以平方级来计算的执行时间。 然而，如果我们用那些不是很直接或者不是很显而易见的分治法，我们能得到比较快的执行结果。 相较而言，才用分治法是个比较好的作法，而归并排序正式充分应用了这个优点。 第二，归并排序可以为你未来的课程做好定位。 在接下来的讨论中，我想我可以帮助你来判断自己的程度是否适合这个课程。 特别要指出的是，在讲解归并算法时，我不会一行一行地解释程序的运行原理，你需要自己理解并编写。 我假设你是已经有了一定经验的程序设计者，你应该能够理解较抽象的算法， 而且应当能自己将算法的原理加以诠释，并能用你熟悉的程序语言来编写出来。 我不知道用这种方式来分析归并排序对你来说是否会很困难， 但是我希望你终将会发现这反而是相对直接的方式， 毕竟，随着课程深入，后面讲到的算法及其分析都会比归并排序要复杂。 所以，换句话说，我们或许可以用归并排序来做个热身。 第三个原因，是因为分析算法的方式和分析其他事物不太一样。 当我们在分析时，我必须先做几个假设性前提，并且集中分析最差情况， 如果不这么做的话，我们就得必须逐项分析所以输入参数的情形并检验每种情况。 然后采用所谓的“渐进分析法”来观察算法效率的增长率，但是这样一来，我们就无法用低阶因子或者常数来表示。 最后，我们会采用所谓的“递归树”的方法来分析归并排序。 这种方法可以用来计算一种算法总共执行了多少个步骤。 就像我们接下来会看到的那样，这个递归树的方法做得不错， 我们可以用它来分析许多种不太的递归和分治型算法，包括我们之前讲过的两数相乘的方法。 这些就是我用归并排序作为开场的理由。 那么，到底归并排序是用来解决那种计算问题呢？ 好吧，先假设你知道什么叫做排序。但是，让我先阐明几件事，以确保我们的认知没有差别。 现在，假设我们有一个含有N个元素的随机数列作为输入，我们要把它们从小到达进行排序后输出。 例如，我们假设这个是输入数列，我们的目的是把它排列成如下所示的输出数列。 让我注解一下：请注意这个输入数列，它总共有8个元素，每个都是从1到8的不重复的整数。 如果个别数有重复的话，这并不会提高困难度；相反，这样反而会更简单一些。 但是为了简化课程的讨论过程，我还是使用不重复的数值。 至于重复数值是否会让归并排序的分析和结果变得不一样这个问题，我就把它留作给同学们的课后作业吧。 所以为了简化起见，我们还是继续使用不重复的数值来作为这个例子。 在我把归并排序的伪代码写下来之前，我先画这个图来表示该算法的过程。 至于程序的部分，即使我只举出一个例子，我想它也应该很容易地写出来。 现在，我们把上一张中得那串数字来作为未排序的输入数列。 由于归并排序是中递归算法，即把原来的问题分解成几个较小的问题，再递归地交给同一个程序，对吧？ 归并排序的本质就是把输入数列进行排序，它会把输入数列分解，再把分解后的较小数列返回给自己。 这是个典型的分治型应用范例。我们把输入数列分成两半，先递归地解决左半部分，再解决有半部分，接着整合出结果。 现在我们来看看图解。 第一个递归取左半部分4个元素作为输入，也就是5,4,1,8；另一个递归取另外四个元素，也就是7,2,6,3。 你可以想象一下，在把分解后的元素输入给递归之前，它们的数字都已经被复制到新的数列里面了。 借由这个神奇的递归，或许你比较喜欢叫做“归纳原理”，我们只要触发那些递归就能把事情做好。 它们会把这两个含有4个元素的数列进行排序，然后把排序完成后的数列返回。 所以，在第一个递归中，我们得到排好序的输出数列是1,4,5,8；同理，第二个递归中得到2,3,6,7。 要完成整个归并排序，就是把以递归方法得到的两个含有4个元素的数列进行组合，成为最后排好序的8个元素。 因此，我们把上面这个步骤称之为“合并” (Merge)。 我希望你已经在思考如何把这个合并过程以有效的计算机程序来实现。 不过我还有一些细节没有交代，我会在之后明确地讲清楚。 事实上，我们就是照着图中的箭头向下走，把数列分解、复制，再把排好序的结果输出，如此而已。 不过我会再深入地讲解，大概需要1-2张幻灯片。 以上就是我用画图的方式讲解的归并排序—— 把原始数列对半分解、以递归的方式分别解决，再使用某种巧妙的程序 把结果合并以得到最好排好序的输出数列。

# 1-6-Merge Sort Pseudocode

0:00 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. 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. 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. 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. 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. 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. 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. 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. 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? 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. 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. 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. 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. 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. 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. 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. 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? 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 symmeterically for the case where the current position in B has the smaller element. 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.

# 1-6-chs

0:00 好的，让我们继续，讨论归并排序的伪代码应该是怎样 首先，我先解释一下什么是伪代码 先不管归并的子程序具体是怎么实现的，假设子程序已经存在 这样，上层的合并就非常简明了。因此，这里会有两个递归调用 然后会有一个合并的步骤。当然，这里我省略了一些细节 因为我有点马虎。不过，就像我之前说的，我不会直接给你 算法程序。尽管也非常接近了。那么我会省略掉哪些东西呢 [这里听不清]首先，在递归算法中 你必须有一个基准。你必须知道 当输入为什么的时候，这个算法就简化到可以停止了 只需要返回一个显然的答案。在排序问题中，这个基准就是 你的数列中只含有0个或1个元素。那么这肯定就是排好序的 你也不能对这个数列再做什么，只需要原样返回就好 好的，我要再明确一下，我不会写下任何基准。但是你们如果想要 让这个归并排序算法可以运行，你们必须在程序中给出，要记住这点。 还有其它一些我在伪代码中不会提及的事，比如如果这个数组的长度为奇数怎么办 如果它有九个元素，那么你就不能把数列平均分成两份 你必须稍作处理，比如一边为5个一边为4个元素 还有，我不会给出你们递归排序的具体细节 比如，我不会去讨论在递归调用中你们究竟要如何 把子数列的值返回给函数。这个主要是 依赖于你们的编程语言。所以 我不会去具体讨论这个问题。我真正想要讨论的是那些 超越于任何编程语言的，抽象出来的算法的概念。 好的，下面来讨论一个算法中较难的部分 你怎么来实现归并的深度？当递归调用做完了它们的工作后 我们有两个已经排好序的子数列。左数列和右数列， 那么我们怎么把这两个数列合并？其实， 上一个幻灯片中，我已经解释过了。思路就是你把输出按序排好 然后用指针去遍历数组，或同时遍历这两个。 我再详细解释一下，这里是归并步骤的伪代码 [一些不明声音]我们先开始介绍 这里一些符号的含义，我们用C 来定义我们的输出数组，也就是我们想要去通过两个排好序的子数组合并得到的。 然后，我用a和b来定义两个递归调用的结果。 第一个递归可以得到a数组 也是输入数组的左半边子数组 b就代表了已经排好序的右半子数组 如之前所说，我们将同时遍历这两个已经排好序的子数组a和 因此，这里需要一个计数器i，来遍历 j来遍历b。i和j的初始值都为1.指向这两个数组的首位 现在我们要做的是，把输出的数值 复制为一个升序的序列。也就是始终把 两个子数组的最小值复制过去。这里你需要想到的 一点是，这个最小元素 一定是在a和b两个子数组的 最开始的位置(除去那些已经写入输出矩阵的元素) 不管是在a数组还是b数组中 整个原始数组的最小值不是在a数组 的头部，就是在b数组的头部。因此你只需要检查这两个位置 然后把较小的数值复制过去 这里的K的目的主要是为了得到输出矩阵当前遍历到的位置。这也是 我们要输出的顺序。再来看一下位置 还有第二个矩阵的位置j。我们现在就得到了， 在两个子数组中已经遍历到的深度。 然后继续比较i和j位置的数值大小，再把较小的复制到C的第k位中 也就是比如a(i)小于b(j)，那我们就把a(i)的元素复制过去 当然，这里要记住把i递增。这样我们就探索完了a数组的第一个数值 同时如果是b(j)中的元素更小，可以用同样的方式进行 这里我不再详细说明了。以免我们过于专注在排序的小细节上而忽略了整个森林 我这里也不会去说明结束条件，因此如果你真的想要 实现这个算法，你必须自己把停止条件加进去。来防止无休止的循环下去 无论对a还是b，你都要去检查i和j是否已经指向了数组的最后 或是判断什么时候你已经把所有的元素都复制到C中了 我会给你一个简洁的伪代码版本 这样你就不用忍受我毫无章法的手写体了 而这也是我们在上一个幻灯片中提过的相同的事情 也就是归并排序的伪代码。好的，这就是归并排序算法 现在，让我们进入这个课题最有趣的部分 归并排序能够获得一个排好序的数组。但是我们为什么会认为它比那些 没有用分治算法思想去排序的方法要好呢？比如，插入排序 换句话说，排序算法的运行时间是多少？现在我会给你一个 准确的定义，就是关于我这里所说的运行时间 稍后会解释，我们这么做是有足够的理论依据的。但是，先从直观上来看 你应该考虑一个算法的运行时间，应该从一个调试者的角度 来考虑算法的运行。每一次你点击运行，然后通过调试器 来一步一步的运行。那么基本上，运行时间就是 你所执行的操作的数量，代码执行过的行数 所以现在的问题就是，你需要点击多少次运行来使程序执行完成 也就是，我们感兴趣的是在归并算法中，当输入数组有n个数字 那么你实际执行的代码行数是多少。当然，这是 一个相当复杂的问题。所以我们先从一个相对简单的方面开始， 先不考虑归并排序实际执行的操作的数量， 不管这个疯狂的递归算法，到底不断重复又重复的调用了它自己多少次 我们先来考虑只是把两个已经排好序的子数组合并 所需要的操作步数。这个任务听起来容易开始多了 我们回忆一下，归并子程序的伪代码 我们现在顺着步骤走，计算这里会用到的 操作步数有多少。这里是初始化步骤。 这两个初始化步骤将各自消耗我们一步操作 因此这里需要2步，来让i=1，j=1.然后我们进入for循环 要知道for循环总共执行的步数，首先需要知道每一步迭代 花费了多少步，这里有一步，然后我们需要比较 A(i)和B(j)，而且每比较完一次，我们就需要做 另两步操作，我们在这或这做一个赋值。然后 要对相关的变量做一个递增。因此每一步迭代将会花费 3步操作。然后还有K递增的步骤，我们可以 把这称为第四步。因此对于这个for循环的N次迭代， 每一次迭代将会执行四次操作。把它们加到一起 我们就得到了这个合并的运行时间。我们再来看一下结果 这个结果是合并的子程序的运行时间，给定一个 M个数字的数组，最多会执行4\*M+2步操作。这里先说明两点 首先，我把数组的维度改了，以免你会困惑 上一页PPT中我们一直在考虑输入N个数字的情况。并且我们也讨论完成了 因此这里我把变量名称换成M。这样当我们考虑合并时会更方便 因为程序是递归的在两个更小的问题上进行的。 不过无论是M还是N，都是一样的。总之对于一个M维的输入数组，我们会执行4\*M+2次操作 另一件事就是，关于我们刚刚究竟是怎样计算代码行数的还有一些不太清楚 的地方。你可能对循环会有异议，就是 每一次循环迭代应该算做两次操作，而不是一次 因为你不仅仅是递增K，同时还需要 和N对比。所以是5\*M+2步而不是4\*M+2 因此计算方式的不同会使结果有一些小差异。不过执行的代码行数 具体是多少并没有太大关系，我们之后会说为什么。 我们暂时还是设定，对于M维的数组， 合并所需要的步骤是4M+2.然后，我们可以放大一下 加上一个虽然看起来很草率，但是却绝对正确的不等式 因为，这会让我们接下来的分析容易很多。4M+2中的2 始终让我感觉很不舒服。我们知道M是大于等于1的 所以可以令这个式子小于6M。这个不等式是肯定成立的 虽然显得很马虎。因为当M很大时，这两个式子也会 相差很大。但是这一切都是为了以后的分析简便考虑 好的，我不期望你们现在能够理解这种很粗暴的将 合并子程序执行所需要的步骤上限提升的方式 但要记住我们的最终目的是算出 将输入数组排序所需要的步骤，而不仅仅是这个子程序 事实上，分析归并排序的主程序会更复杂一些 因为它在不停的回调自己本身，而我们需要去分析的这个 递归调用的次数是个呈指数级增长的过程 我们现在还要知道一件事 即每次进行递归调用时，输入数组会不断的减小 每次都是之前的一半大小 因此对于这个问题，一方面是子问题的分裂造成的膨胀， 而另一面又是子问题会越来越小 二者之间形成了牵制 要解决这二者之间的矛盾就要取决于是什么在驱动我们的合并排序分析 我将会完整的分析给你们看 究竟归并排序需要执行的代码行数有多少，并且还可以计算出 一个精确的上界。因此，这里我们可以先给出一个结论，接下来会去证明，结论就是 归并排序总共的步骤不超过 6NlogN+6N 这些就是要把一个N维的数组排序所需要的步数 我们先来讨论一下，这个结果是不是足够好，是不是比其它的方法更优 我们已经知道这个式子是归并排序执行次数的上界，所以答案是肯定的 并且它体现出了分治算法的优势，回顾一下 在一些更简单的排序方法中，比如之前提过的插入排序，选择排序还有 冒泡排序，它们的复杂性是输入数组大小的二次函数 也就是说它们所需要的时间是 N的评分，而归并排序需要的时间 最多为NlogN。 这样你应该能够感觉出来 哪种方式更优，因此，想在这里再次提醒那些 仍然对算法心生恐惧或是感觉无从入手的学生，到底什么是算法。 你应该从以下的角度去考虑算法，比如你有一个X轴 这上面有N个点，从1到无穷 为了方便比较，我们只考虑恒等函数 比如函数是F(n)=n，然后让我们把算法跟它来对比 那么什么是算法，在这里，我们可以这么来看， 以2为底的logn，就是你在你的计算器中输入N，然后你令它除2，然后 不停的重复这个过程 然后你可以 数一下需要重复多少次这个过程，最终才可以得到 一个小于一的数。比如对于32，你需要除5次 来得到小于等于1的数。因此log32是5.输入1024 你需要除2除10次，那么log1024等于10，等等 那么对于1000呢，通过上面的例子，你应该能够看出来 也差不多需要除10次。所以这个算法需要的输入比起原始的N已经小很多很多了 从图形上，这个算法会是长这个样子， 首先会是一条曲线，然后随着N的增大迅速的变平。这个新的F(n)就是 以2为底的logn。我希望你们有时候也可以画一下 可以通过电脑或是一些图形软件画的更精确一点 但是对数函数运行起来会比恒等函数慢很多，因此 总之结果就是，一个运行时间近似于nlogn的倍数的排序算法 比起那些运行时间近似于n的平方的倍数的排序算法 要快很多。尤其是n很大的时候

# 1-7-Merge Sort Analysis

0:00 In this video we'll be giving a running time analysis of the merge sort algorithm. In particular, we'll be substantiating the claim that the recursive divide and conquer merge sort algorithm is better, has better performance than simple sorting algorithms that you might know, like insertion sort, selection sort and bubble sort. So in particular, the goal of this lecture will be to, mathematically argue the following claim, from an earlier video. That, in order to sort an array of n numbers, the merge sort algorithm needs no more than a constant times n log in operations. That's the maximum number of lines of code it will ever execute. Specifically 6 times n log n + 6n operations. So how are we going to prove this claim? We're going to use what is called a recursion tree method. The idea of the recursion tree method is to write out all over the work done by the recursive merge sort algorithm in a tree structure with the children of a given node corresponding to the recursive calls made by that node. The point of this tree structure is it will facilitate a interesting way to count up the overall work done by the algorithm and will greatly facilitate the analysis. So specifically what is this tree? So at level zero we have a root. 1:14 And this corresponds to the outer call of merge sort, okay so I'm going to call this level zero. Now, this tree is going to be binary in recognition of the fact that each invocation of MergeSort makes two recursive calls. So the two children will correspond to the two recursive calls of MergeSort. So, at the root we operate on the entire input array, so let me draw a big array indicating that, and at level one we have one sub problem for the left half and another sub problem for the right half of the input array. And I'll call these first two recursive calls level one. Now, of course, each of these two level one recursive calls will themselves make two recursive calls. Each operating on then a quarter of the original input array. So those are level two recursive calls of which there are four. And this process will continue until eventually the recursion bottoms out in base cases when there's only an array of size zero or one. So now I have a question for you which I'll give you in the form of a quiz. Which is at the bottom of this recursion tree corresponding to the base cases, what is the level number at the bottom? So at what level do the leaves in this tree reside? 2:22 Okay, so hopefully you guessed, correctly guessed that the answer is the second one. So namely that the number of levels of the recursion tree is essentially logarithmic in the size of the input array. The reason is basically that the input size is being decreased by a factor two with each level of the recursion. If you have an input size of n at the outer level, then each of the first set of recursive calls operates on a array of size n over 2. At level two, each array has size n over 4 and so on, whereas if the recursion bottom out, well, down at the base cases, where there's no more recursion, which is where the input array has size one or less. So in other words, the number of levels of recursion is exactly the number of times you need to divide n by 2, until you get down to a number that's most one. And recall, that's exactly the definition of a logarithm base 2 of n. So since the first level is level zero and the last level is level log base 2 of n. The total number of levels is actually log base 2 of n plus 1. And when I write down this expression I'm here assuming that n is a power of 2 which is not a big deal. I mean the analysis is usually extended to the case where n is not a power of 2. But this way we don't have to think about fractions log base 2 of n then is an integer. Okay, so let's return to the recursion tree, and let me just redraw it really quick. 3:39 So again, down here at the bottom of the tree, we have the leaves. And i.e., the base cases, where there's no more recursion. Which, when n is a power of 2, correspond exactly to single element arrays. 3:51 So that's the recursion tree corresponding to an invocation of MergeSort. And the motivation for writing down, for organizing the work performed by MergeSort in this way, is it allows us to count up the work level by level. And we'll see that's a particularly convenient way to account for all of the different lines of code that get executed. Now to see that in more detail, I need to ask you identify a particular pattern. So first of all, the first question is, at a given level j of this recursion, exactly how many distinct sub problems are there as a function of the level j. That's the first question. The second question is, for each of those distinct sub problems at level j, what is the input size. So what is the size of the array which is passed to a subproblem residing at level j of this recursion tree? 4:39 So the correct answer is the third one. 4:42 So first of all at a given level j, there's precisely 2 to the j distinct subproblems. There is one outermost subproblem at level zero. It has two recursive calls. Those are the subproblems at level one, and so on. In general, since MergeSort calls itself twice, the number of subproblems is doubling each level, so that gives us the expression 2 to the j for the number of subproblems at level j. 5:05 On the other hand via a similar argument the input size is halving each time with each recursive call you pass at half of the input that you were given. So at each level of the recursion tree we're seeing half of the input size of the previous level. So after j levels since we started with an input size n after j levels each subproblem will be operating on an array of length n over 2 the j. Okay so now let's put this pattern to use, and actually count up all of the lines of code, and thenMergeSort executes. And as I said before, the key idea is to count up the work level by level. Now to be clear, when I talk about the amount of work done at level j. What I'm talking about is the work done by those 2 to the j invocations of MergeSort not counting their respective recursive calls, not counting work which is going to get done in the recursion lower in the tree. Now recall merge sort is a very simple algorithm, it just three lines of code, first there's a recursive call so we're not counting that, second, there's another recursive call. Again, we're not counting that at level j. And then third, we just invoke the merge subroutine. So really outside the recursive cause all that MergeSort does is a single indication of merge. Further, recall we already have a good understanding of the number of lines of code that merge needs. On an input of size m, it's going to use at most 6m lines of code. That's an analysis that we did in the previous video. So let's fix a level j. We know how many subproblems there are, 2 to the j. We know the size of each subproblem, n over 2 to the j. And we know how much work merge needs on such an input. We just multiply by 6. And then we just multiply it out. And we get the amount of work done at a level j. Okay at all of the little j subproblems, so here it is in more detail. 6:46 All right so, we start with just the number of different subproblems at level j and we just noticed that that was at most 2 to the j. 6:54 We also observed that each level j subproblem is past an array as input which has length n over 2 to the j. And we know that the merge subroutine, when given an array of size n over 2 to the j, will execute at most 6 times that many number of lines of code. So to compute the total amount of work done at level j, we just multiply the number of problems times the work done per subproblem. And then something sort of remarkable happens. We get this cancellation of the two 2 to the j's and we get an upper bound 6n which is independent of the level j. So we do at most 6n operations at the root. We do at most 6n operations of level one, at level two, and so on okay. It's independent of the level. Morally, the reason this is happening because of a perfect equilibrium between two competing forces. First of all, the number of subproblems is doubling with each level of the recursion tree. But secondly, the amount of work that we do per subproblem is halving with each level of the recursion trees. And so those two cancel out. We get an upperbound 6n, which is independent of the level j. Now, here's why that's so cool, right, we don't really care about the amount of work just at a given level. We care about the amount of work that MergeSort does ever at any level. But if we have a bound on the amount of work at a level which is independent of the level, then our overall bound is really easy. What do we do? We just take the number of levels. And we know what that is. It's exactly log base 2 of n + 1. Remember, the levels are zero through log base 2 of n inclusive. And then we have an upper bound 6n for each of those log n plus 1 levels. So, if we expand out this quantity, we get exactly the upper bound that was claimed earlier, namely that the number of operations MergeSort executes is at most 6n times log based 2 of n plus 6n. So that my friends, is a running time analysis of the merge sort algorithm. That's why its running time is bounded by a constant times nlogn, which especially has n grows large, it is far superior to the more simple iterative algorithms like insertion or selection sort.

# 1-8-Guiding Principles for Analysis of Algorithms

0:00 Having completed our first analysis of an algorithm, namely an upper bound on the running time of the Merge Short algorithm. What I wanna do next is take a step back, and be explicit about three assumptions, three biases that we made when we did this analysis of Merge Short, and interpreted the results. These three assumptions we will adopt as guiding principles for how to reason about algorithms, and how to define a so called fast algorithm for the rest of the course. So, the first guiding principle is that we used what's often called worst case analysis. By worst case. Analysis, I simply mean that our upper bound of six N log N plus six N. Applies to the number of lines of executed for every single input array of length end. We made absolutely no assumptions about the input, where it comes from, what it looks like beyond what the input length N was. Put differently, if, hypothetically, we had some adversary whose sole purpose in life was to concoct some malevolent input designed to make our algorithm run as slow as possible. The worst this adversary could do, is. Upper bounded by this same number, 6N log N + 6N. Now, this, so, sort of worst case guarantee popped out so naturally from our analysis of Merge Short, you might well be wondering, what else could you do? Well, two other methods of analysis, which do have their place, although we won't really dicuss them in this course, are quote unquote, average case analysis. And also the use of a set of prespecified benchmarks. By average case analysis, I mean, you analyze the average running time of an algorithm under some assumption about the relative frequencies of different inputs. So, for example, in the sorting problem, one thing you could do, although it's not what we do here. You could assume that every possible input array is equally unlikely, and then analyze the average running time of an algorithm. By benchmarks, I just mean that one agrees up front about some set, say ten or twenty, Benchmark inputs, which are thought to represent practical or typical inputs for the algorithm. Now, both average-case analysis and benchmarks are useful in certain settings, but for them to make sense, you really have to have domain knowledge about your problem. You need to have some understanding of what inputs are more common than others, what inputs better represent typical inputs than others. By contrast, in worst-case analysis, by definition you're making absolutely no assumptions about where the input comes from. So, as a result, worst-case analysis is particularly appropriate for general-purpose sub-routines. Sub-routines that you design. Find without having any knowledge of how they will be used or what kind of inputs they will be used on. And happily, another bonus of doing worst case analysis, as we will in this course, it's usually mathematically much more attractable than trying to analyze the average performance of an algorithm under some distribution over inputs. Or to understand the detailed behavior of an algorithm on a particular set of benchmark inputs. This mathemetical tractabilty was reflected in our Merge Sort analysis, where we had no a priori goal of analyzing the worst case, per se. But it's naturally what popped out of our reasoning about the algorithm's running time. The second and third guiding principles are closely related. The second one is that, in this course, when we analyze algorithms, we won't worry unduly about small constant factors or lower order terms. We saw this philosophy at work very early on in our analysis of merge sort. When we discussed the number of lines of code that the merge subroutine requires. We first upper-bounded it by 4m plus two, for an array of length m, and then we said, eh, let's just think about it as 6m instead. Let's have a simpler, sloppy upper-bound and work with that. So, that was already an example of not worrying about small changes in the constant factor. Now, the question you should be wondering about is, why do we do this, and can we really get away with it? So let me tell you about the justifications for this guiding principle. So the first motivation is clear, and we used it already in our merge short analysis. Which is simply way easier mathematically, if we don't have to, precisely pin down what the [inaudible] constant factors and lower-order terms are. The second justification is a little less obvious, but is extremely important. So, I claim that, given the level at which we're describing and analyzing algorithms in this course, it would be totally inappropriate to obsess unduly about exactly what the constant factors are. Recall our discussion of the merge subroutine. So, we wrote that subroutine down in pseudocode, and we gave it analysis of 4m plus two on the number of lines of code executed, given an input of length m. We also noted that, it was somewhat ambiguous exactly how many lines of code we should count it as, depending on how you count loop increments and so on. So even there it's small constant factors could creep in given the under specification of the pseudo code. Depending on how that pseudo code gets translated into an actual program language like C or Java. You'll see the number of lines of code deviate even further, not but a lot but again by small constant factors. When such a program is then compiled down into machine code, you'll see even greater variance depending on the exact processor, the compiler, the compiler optimizations, the programming implementation, and so on. So to summarize, because we're going to describe algorithms at a level. That transcends any particular programming language. It would be inappropriate to specify precise constants. The precise constants were ultimately determined by more machine dependent aspects like who the programmer is, what the compiler is, what the processor is, and so on. And now the third justification is frankly, we're just going to be able to get away with it. [sound] That is, one might be concerned that ignoring things like small constant factors leads us astray. That we wind up deriving results which suggest that an algorithm is fast when it's really slow in practice, or vice versa. And for the problems we discuss in this course we'll get extremely accurate predictive power. Even though we won't be keeping track of lower terms and constant factors. When the mathematical analysis we do suggests that an algorithm is fast, indeed it will be. When it suggests that it's not fast, indeed that will be the case. So we lose a little bit of granularity of information. But we don't lose in what we really care about, which is accurate guidance about what algorithms are gonna be faster than others. So the first two justifications, I think, are pretty self evident. This third justification is more of an assertion, but it's one we'll be baking up over and over again as we proceed through this course. Now, don't get me wrong. I'm not saying constant factors aren't important in practice. Obviously, for crucial programs the constant factors are hugely important. If you're running the sorta crucial loop, you know, your start up's survival depends on, by all means optimize the constant like crazy. The point is just that understanding tiny constant factors in the analysis is an inappropriate level of granularity for the kind of algorithm analysis we're going to be doing in this course. Okay, lets move on the, the third and final guiding principle. So the third principle is that we're going to use what's called asymptotic analysis, by which I mean we will focus on the case of a large input sizes. The performance of an algorithm as the size N of the input grows large, that is, tends to infinity. Now this focus on large input size is it was already evident when we interpreted our bound on Merge Sort. So, how did we describe the bound on Merge Sort? We said, oh, well, it needs a number of operations proportional, a constant fact or times in login. And we very cavalierly declared that this was better than any algorithm which has quadratic dependence of it's running time on the number of operations. So for example, we argued that merge sort is a better, faster algorithm than something like insertion sort, without actually discussing the constant factors at all. So mathematically. We were saying the running time of merge short, which we know, which we can represent as the function. Six N log base two of N + 6N is better than any function which has a quadratic dependence on n. Even one with a small constant like lets say 1/2 N squared which might roughly be the running time of insertion sort. And this is a mathematical statement that is true if and only if N is sufficiently large once N grows large it is certainly true that the expression on the left is smaller than the expression on the right but for small N the expression on the right is actually going to be smaller because of the smaller leading term so in saying that merge sort is superior to insertion sort the bias is that we're focusing on problems with a large N so the question you should have is that reasonable is that a justified assumption to focus on large input sizes and the answer is certainly yes. So the reason we focus on large input sizes is because, frankly, those are the only problems which are even, which are at all interesting. If all you need to do is sort 100 numbers, use whatever method you want, and it's gonna happen instantaneously on modern computers. You don't need to know say, the divide and conquer paradigm, if all you need to do is sort 100 numbers. So one thing you might be wondering is if, with computers getting faster all the time according to Moore's Law, if really, it doesn't even matter to think about algorithmic analysis, if eventually all problem sizes will just be trivially solvable on super fast computers. But, in fact, the opposite is true. Moore's Law, with computers getting faster, actually says that our computational ambitions will naturally grow. We naturally focus on ever larger problem sizes. And the gulf between an N squared algorithm and an m log n algorithm will become ever wider. A different way to think about it is in terms of how much bigger a problem size you can solve. As computers get faster. If you are using an algorithm with a running time which is proportional to the input size then the computers get faster by a factor of four then you can solve problems that are factor of four or larger. Whereas if you are using an algorithm whose running time is proportional to the square of the input size then a computer gets faster by a factor of four, you can only solve double the problem size and we'll see even starker examples of this gulf between different algorithm approaches as the time goes on. So to drive this point home. Let me show you a couple of graphs. So what we're looking at here, is we're looking at a graph, of two functions. So the solid function. Is the upper bound that we proved on merge sort. So this is gonna be 6nlog(base2)n plus 6n. And the dotted line is an estimate. A rather generous estimate about the running time of, [inaudible] sort. Namely one-half times N. Squared. And we see here in the graph exactly the behavior that we discussed earlier, which is that the small N. Down here. In fact because one-half N. Squared has a smaller leading constant it's actually a smaller function. And this is true up to this crossing point of maybe 90 or so. Again, beyond n=90. The quadratic growth in the N squared term. Overwhelms the fact that it had a smaller constant and it starts being bigger than this other function six of N + six N so in the regime below 90 it's predicting that the insertion store will be better and in the regime above 90 it's predicting that merge sort will be faster. Now here's what's interesting let's scale the X axis let's look well beyond this crossing point of 90 let's just increase it in order of magnitude up to a raise in size 1500. And I want to emphasize these are still very small problem sizes. If all you need to do is sort arrays of size 1500 you really don't need to know Divide-and-conquer or anything else we'll talk about -- that's a pretty trivial problem on modern computers. [sound]. So what we're seeing is, that even for very modest problem sizes here, array of, of, size, say 1500. The quadratic dependence in the insertion sort bound is more than dwarfing the fact, that it had a lower constant factor. So in this large regime, the gulf between the two algorithms is growing. And of course, if I increased it another 10X or 100x or 1000x to get to genuinely interesting problem sizes, the gap between these two algorithms would be even bigger, it would be huge. That said, I'm not saying you should be completely ignorant of constant factors when you implement algorithms. It's still good to have a general sense of what these constant factors are so for example in highly tuned versions of Merge Sort which you'll find in mny programming libraries. In fact, because of the difference in constant factors, the algorithm will actually switch from Merge Sort over to insertion sort, once the problem size drops below some particular threshold, say seven elements, or something like that. So for small problem sizes, you use the algorithm with smaller constant factors, and the insertion sort for larger problem sizes, you use the algorithm and better rate of growth, mainly merge short. So, to review our first guiding principal is that we're going to pursue worse case analysis. We're going to look to bounds on the performance, on the running time of an algorithm which make no domain assumptions, which make no assumptions about which input of a given length the algorithm is provided. The second guiding principal is we're not going to focus on constant factors or lower returns, that would be inappropriate, given the level of granularity at which we're describing algorithms and third is where going to focus on the rate of growth of algorithms for large problem sizes. Putting these three principles together, we get a mathematical definition of a fast algorithm. Namely, we're gonna pursue algorithms whose worst case running time grows slowly as a function of the input size. So let me tell you how you should interpret what I just wrote down in this box. So on the left hand side is clearly what we want. Okay, we want algorithms which run quickly if we implement them. And on the right hand side is a proposed mathematical surrogate of a fast algorithm. Right, the left hand side is not a mathematical definition. The right hand side is, as well become clear in the next set of lectures. So we're identifying fast algorithms, which, those that have good asymptotic running time which grows slowly with the input size. Now, what would we want from a mathematical definition? We'd want a sweet spot. On one hand we want something we can actually reason about. This is why we zoom out and squint and ignore things like constant factors and lower terms. We can't keep track of everything. Otherwise we'd never be able to analyze stuff. On the other hand we don't want to throw out the baby with the bath water, we want to retain predictive power and this turns out this definition turns out for the problems we're going to talk about in this course, to be the sweet spot for reasoning about algorithms okay worst case analysis using the asymptotic running time. We'll be able to prove lots of theorems. We'll be able to establish a lot of performance guarantees for fundamental algorithms but at the same time we'll have good predictive power what the theory advocates will in fact be algorithms that are known to be best in practice. So, the final explanation I owe you, is, what do I mean by, the running time grows slowly, with respect to the input size? Well, the answer depends a little bit on the context, but, for almost all of the problems we're going to discuss, the holy grail will be to have what's called a linear time algorithm, an algorithm whose number of instructions grows proportional to the input size. So, we won't always be able to achieve linear time, but that's, in some sense, the best case scenario. Notice, linear time is even better than what we achieved with our merge short algorithm for sorting. Merge short runs a little bit superlinear, it's n times log n, running as the input size. If possible, we. To be linear time. It's not always gonna be possible, but that is what we will aspire toward. For most of the problems we'll discuss in this course. Looking ahead, the next series of videos is going to have two goals. First of all, on the analysis side, I'll describe formally what I mean by asymptotic running time. I'll introduce "Big Oh" notation and its variants, explain its mathematical definitions, and give a number of examples. On the design side, we'll get more experience applying the divide and conquer paradigm to further problems. See you then.

# 2-1-The Gist

0:02 In this sequence of lectures, we're going to learn Asymptotic Analysis. This is the language, by which every serious computer programmer and computer scientist, discusses, the high level performance, of computer algorithms. As such, it's a totally, crucial, topic. In this video, the plan is to segueway between the high level discussion that you've already seen in the course introduction. And the mathematical formalism which we're going to start developing in the next video. Before getting into that mathematical formalism, however, I want to make sure that the topic is well motivated. That you have solid intuition for what it's trying to accomplish, and also that you've seen a couple of simple, intuitive examples. Let's get started. 0:44 Asymptotic analysis provides basic vocabulary for discussing the design and analysis of algorithms. And while it is a mathematical concept, it is by no means math for math's sake. You will very frequently hear serious programmers saying that such and such code runs in o of n time where such and such other code runs in o of n squared time. It's important you know what programmers mean when they make statements like that. 1:09 The reason this vocabulary is so ubiquitous is that it identifies a sweet spot for discussing the high level performance of algorithms. What I mean by that, is it is on the one hand, coarse enough, to suppress all of the details that you want to ignore. Details that depend on the choice of architecture, the choice of programming language, the choice of compiler, and so on. 1:34 On the other hand it's sharp enough to be useful. In particular, to make predictive comparisons between different high level algorithmic approaches to solving a common problem. This is going to be especially true for large inputs. And remember as we discussed in some sense, large inputs are the interesting ones, those are the ones for which we need algorithmic ingenuity. For example, asymptotic analysis will allow us to differentiate between better and worse approaches to sorting. Better and worse approaches to multiplying two integers and so on. 2:10 Now most serious programmers if you ask them, what's the deal with asymptotic analysis anyways? They'll tell you reasonably, that the main point is to suppress both leading constant factors and lower order terms. 2:24 Now, as we'll see, there's more asymptotic analysis than just these seven words here. But long term, ten years from now if you only remember seven words about asymptotic analysis, I'll be reasonably happy if these are the seven words that you remember. So how do we justify adopting a formalism which essentially by definition, suppresses constant factors and lower order terms? Well lower order terms basically, by definition, become increasingly irrelevant as you focus on large inputs. Which, as we've argued, are the interesting inputs, the ones where algorithmic ingenuity is important. As far as constant factors, these are going to be highly dependent on the details of the environment, the compiler, the language and so on. So if we want to ignore those details, it makes sense to have a formalism which doesn't focus unduly on leading constant factors. 3:14 Here's an example. Remember when we analyzed the merge sort algorithm, we gave an upper bound on its running time that was 6 times n log n plus 6n. Where n was the input length, the number of numbers in the input array. So the lower order term here is the 6n. That's growing more slowly than than n log n, so we just drop that. And then the leading constant factor is the 6 so we suppress that as well. After those two suppressions we're left with a much simpler expression, n log n. 3:44 The terminology would then be to say that the running time of merge sort is big O of n log n. So in other words, when you say that an algorithm's running time is big O of some function. What you mean is that after you've dropped the lower order terms, and suppressed the leading constant factor, you're left with the function f of n. Intuitively, that is what the big O notation means. So to be clear, I'm certainly not asserting thay constant factors never matter when you're designing and analyzing algorithms. Rather, I'm just saying that when you think about high level algorithmic approaches, when you might want to make a comparison between fundamentally different ways of solving a problem. Asymptotic analysis is often the right tool for giving you guidance about which one is going to perform better. Especially on reasonably large inputs. Now, once you've committed to a particularly algorithmic solution to a problem. Of course, you might want to then work harder to improve the leading constant factor, perhaps even to improve the lower order terms. By all means if the future of your start up depends on how efficiently you implement some particular set of lines of code, have at it. Make it as fast as you can. In the rest of this video I want to go through four very simple examples. In fact these examples are so simple, if you have any experience with big O notation, you're probably better off just skipping the rest of this video. And moving on to the mathematical formalism that we begin in the next video. But if you've never seen it before, I hope these simple examples will get you oriented. 5:17 So let's begin with a very basic problem, searching an array for a given integer. 5:26 Let's analyze the straight forward algorithm for this problem where we just do a linear scan through the array. Checking each entry to see if it is the desired integer t. 5:37 That is the code just checks each array entry in turn. If it ever finds the integer t it returns TRUE. If it falls off the end of the array without finding it, it returns FALSE. So what do you think? We haven't formally defined big O notation, but I've given you an intuitive description. What would you say is the running time of this algorithm as a function of the length of the array, capital A? 6:03 So the answer I'm looking for is C, big O of n. Or equivalently we would say that the running time of this algorithm is linear in the input length n. 6:12 Why is that true? Well let's think about how many operations this piece of code is going to execute. Actually the lines of code executed is going to depend on the input. It depends on whether or not the target t is contained in the array A. And if so, where in the array A it lies. But, in the worst case this code will do an unsuccessful search. T will not be in the array and the code will scan through the entire array A and return FALSE. The number of operations then is a constant. There is some initial setup, perhaps and maybe it's an operation to return this final bullying value. But outside of that constant, which will get suppressed in the big O notation, it does a constant number of operations per entry in the array. And you can argue about what the constant is, if it's two, three, four operations per entry point in the array. But the point is, whatever that constant is two, three, or four, it gets conveniently suppressed by the big O notation. So as a result, the total number of operations would be linear in n, and so the big O notation will just be o of n. 7:17 So that was the first example. In the last three examples, I want to look at different ways that we could have two loops. And in this example, I want to think about one loop followed by another, so two loops in sequence. I want to study almost the same problem as the previous one. Where now we are just given two arrays, capital A and capital B, let's say both are the same length. And we want to know whether the target t is in either one of them. Again, we'll look at the straight forward algorithm that we just searched through a. And if we fail to find t and a, we search through b. If we don't find t and b either, than we have to return return FALSE. 7:52 So the question then is exactly the same as last time given this new, longer piece of code. What in big O notation is it's running time? 8:05 Well, the question was the same and in this case the answer was the same. So this algorithm, just like the last one, has running time big O of n. If we actually count the number of operations of course it won't be exactly the same as last time. It'll be roughly twice as many operations as the previous piece of code. That's because we have to search two different arrays each of link in. So, whatever work we did before we now do it twice as many times. Of course that two being a constant. Independent of the input length n, is going to get suppressed once we passed a big O notation. So this, like the previous algorithm, is a linear time algorithm, it has running time big O of n. Let's look at a more interesting example of two loops, where rather than processing each loop in sequence, they're going to be nested. In particular, let's look at the problem of searching whether two given input arrays each of link n, contain a common number. 9:06 The code that we're going to look at for solving this problem is the most straight forward one you can imagine, where we just compare all possibilities. So for each index i into the array a, each index j into the array b. We just see a if A[i] is the the same number as B[j]. If it is we return TRUE. If we exhaust all of the possibilities without ever finding equal elements, then we're safe in returning a FALSE. 9:30 The question of course is in terms of big O notation asymptotic analysis as a function of the array length n, what is the running time of this piece of code? 9:45 So this time the answer has changed. For this piece of code the running time is not big O with n, but it is big O of n squared. So, we might also called this a quadratic time algorithm. Because the running time is quadratic in the input length n. So, this is one of those kinds of algorithms where if you double the input link. Then the running time of the algorithm will go up by a factor of four rather than by a factor of two, like in the previous two pieces of code. So why is this? Why does it have quadratic running time big O of n squared? Well again, there's some constant setup costs which gets suppressed in the big O notation. Again, for each fixed choice of an entry i into array A and an index j for array B, for each fixed choice of i and j, we only do a constant number of operations. The particular constants are relevant because it gets suppressed in the big O notation. What's difference is that there's a total of n squared iterations of this double four loop. In the first example we only had n iterations of a single four loop. In our second example because one four loop completed before the second one began we had only 2n iterations overall. Here, for each of the n iterations of the outer four loop, we do n iterations of the inner four loop. So that gives us the n times n, ie n squared total iterations. So that's going to be the running time of this piece of code. 11:08 Lets wrap up with one final example, it will again be nested four loops. But this time we're going to be looking for duplicates in a single array A. Rather than needing to compare two distinct arrays A and B. 11:21 So here's the piece of code we're going to analyze for solving this problem, for detecting whether or not the input array A has duplicate entries. There's only two small differences relative to the code that we went through on the previous slide when we had two different arrays. 11:36 The first change won't surprise you at all which is instead of referencing the array B, I change that B to an A. All right, so I'm just going to compare the i th entry of A with the j th entry of A. The second change is a little bit more subtle. Which is I changed the inner for loops so that the index j begins at i plus one. Where i is the current value of the outer four loop's index, rather than starting at the index one. I could have had it start at the index one, that would still be correct, but it would be wasteful and you should think about why. If we started the inner four loops index at one, then this code would actually compare each distinct pair of elements of A to each other twice. Which of course is silly. You only need to compare two different elements of A to each other once, to know whether they're equal or not. So this is the piece of code, the question is the same as it always is. What in terms of big O notation and the input link n is the running time of this piece of code? 12:35 So the answer to this question, same as the last one, big O of n squared. That is this piece of code is also has quadratic running time. So what I hope was clear was that whatever the running time of this piece of code is. It's proportional to the number of iterations of this double four loop. Like in all the examples, we do constant work per iteration. We don't care about the constant, it gets suppressed by the Big O notation. So all we gotta do is figure out how many iterations there are of this double for loop. My claim is that there's roughly n squared over two iterations of this double four loop. There's a couple ways to see that. Informally we discussed how the difference between this code and the previous one, is that instead of counting something twice, we're counting it once. So that saves us a factor of two in the number of iterations. Of course, this one half factor gets suppressed by the big O notation anyways. So the big O running time doesn't change. A different argument would just say, there's one iteration for every distinct choice of i and j of indices between one and n. And a simple counting argument says that there's n choose two such choices of distinct i and j. Where n choose two is the number n times n minus one over two. And again suppressing lower order terms in the constant factor, we still get a quadratic dependence on the length of the input array A. 13:52 So that wraps up some of the sort of just simple basic examples. I hope this gets you oriented, you have a strong intuitive sense, for what Big O notation is trying to accomplish, and how it's defined mathematically. Let's now move on to both the mathematical development and some more interesting algorithms.

# 2-2-Big-Oh Notation

0:00 In the following series of videos, we'll give a formal treatment of asymptotic notation, in particular big-Oh notation, as well as work through a number of examples. Big-Oh notation concerns functions defined on the positive integers, we'll call it T(n) We'll pretty much always have the same semantics for T(n). We're gonna be concerned about the worst-case running time of an algorithm, as a function of the input size, n. So, the question I wanna answer for you in the rest of this video, is, what does it mean when we say a function, T(n), is big-Oh of f(n). Or hear f(n) is some basic function, like for example n log n. So I'll give you a number of answers, a number of ways of, to think about what big-Oh notation really means. For starters let's begin with an English definition. What does it mean for a function to be big-Oh of f(n)? It means eventually, for all sufficiently large values of n, it's bounded above by a constant multiple of f(n). Let's think about it in a couple other ways. So next I'm gonna translate this English definition into picture and then I'll translate it into formal mathematics. So pictorially you can imagine that perhaps we have T(n) denoted by this blue functions here. And perhaps f(n) is denoted by this green function here, which lies below T(n). But when we double f(n), we get a function that eventually crosses T(n) and forevermore is larger than it. So in this event, we would say that T(n) indeed is a Big-Oh of f(n). The reason being that for all sufficiently large n, and once we go far enough out right on this graph, indeed, a constant multiple times of f(n), twice f(n), is an upper bound of T(n). So finally, let me give you a actual mathematical definition that you could use to do formal proofs. So how do we say, in mathematics, that eventually it should be bounded above by a constant multiple of f(n)? We see that there exists two constants, which I'll call c and n0. So that T(n) is no more than c times f(n) for all n that exceed or equal n0. So, the role of these two constants is to quantify what we mean by a constant multiple, and what we mean by sufficiently large, in the English definition. c obviously quantifies the constant multiple of f(n), and n0 is quantifying sufficiently large, that's the threshold beyond which we insist that, c times f(n) is an upper-bound on T(n). So, going back to the picture, what are c and n0? Well, c, of course, is just going to be two. And n0 is the crossing point. So we get to where two f(n). And T(n) cross, and then we drop the acentode. This would be the relative value of n0 in this picture, so that's the formal definition, the way to prove that something's bigger of f(n) you exhibit these two constants c and n0 and it better be the case that for all n at least n0, c times f(n) upper-bounds T(n). One way to think about it if you're trying to establish that something is big-Oh of some function it's like you're playing a game against an opponent and you want to prove that. This inequality here holds and your opponent must show that it doesn't hold for sufficiently large n you have to go first your job is to pick a strategy in the form of a constant c and a constant n0 and your opponent is then allowed to pick any number n larger than n0 so the function is big-Oh of f(n) if and only if you have a winning strategy in this game. If you can up front commit to constants c and n0 so that no matter how big of an n your opponent picks, this inequality holds if you have no winning strategy then it's not big-Oh of f(n) no matter what C and n0 you choose your opponent can always flip this in equality. By choosing a suitable, suitable large value of n. I want to emphasis one last thing which is that these constants, what do I mean by constants. I mean they are independent of n. And so when you apply this definition, and you choose your constant c and n0, it better be that n does not appear anywhere. So C should just be something like a thousand or a million. Some constant independent of n. So those are a bunch of way to think about big-Oh notation. In English, you wanna have it bound above for sufficiently large numbers n. I'm showing you how to translate that into mathematics that give you a pictorial representation. And also sort of a game theoretic way to think about it. Now, let's move on to a video that explores a number of examples.

# 2-3-Basic Examples

0:00 Having slogged through the formal definition of big O notation, I wanna quickly turn to a couple of examples. Now, I wanna warn you up front, these are pretty basic examples. They're not really gonna provide us with any insight that we don't already have. But they serve as a sanity check that the big O notation's doing what its intended purpose is. Namely to supress constant factors and low order terms. Obviously, these simple examples will also give us some, facility with the definition. So the first example's going to be to prove formally the following claim. The claim states that if T(n) is some polynomial of degree "k", so namely a<u>k n^k. Plus all the way up to a<u>1 N + a<u>0. For any integer "k", positive</u></u></u> integer "k" and any coefficients a<u>i's positive or negative. Then: T(n) is big</u> O of n^k. So this claim is a mathematical statement and something we'll be able to prove. As far as, you know, what this claim is saying, it's just saying big O notation really does suppress constant factors and lower order terms. If you have a polynomial then all you have to worry about is what is the highest power in that polynomial and that dominates its growth as "n" goes to infinity. So, recall how one goes about showing that one function is big O of another. The whole key is to find this pair of constants, c and n<u>0, where c quantifies the constant multiple</u> of the function you're trying to prove big O of, and n<u>0 quantifies what you mean</u> by "for all sufficiently large n." Now, for this proof, to keep things very simple to follow, but admittedly a little mysterious, I'm just gonna pull these constants, c and n<u>0, out of a hat. So, I'm not gonna tell you how I derived them,</u> but it'll be easy to check that they work. So let's work with the constants n<u>0</u> equal to one, so it's very simple choice of n<u>0 and then "c" we are gonna pick to</u> be sum of the absolute values of the coefficients. So the absolute value of "a<u>k"</u> plus the absolute value of "a<u>(k-1)", and so on. Remember I didn't assume that</u> the pol..., the original polynomial, had non-negative coefficients. So I claim these constants work, in the sense that we'll be able to prove to that, assert, you know, establish the definition of big O notation. What does that mean? Well we need to show that for all "n" at least one (cause remember we chose n<u>0 equal to</u> one), T(n) (this polynomial up here) is bounded above by "c" times "n^k", where "c" is the way we chose it here, underlined in red. So let's just check why this is true. So, for every positive integer "n" at least one, what do we need to prove? We need to prove T(n) is upper bounded by something else. So we're gonna start on the left hand side with T(n). And now we need a sequence of upper bounds terminating with "c" times "n^k" (our choice of c underlined in red). So T(n) is given as equal to this polynomial underlined in green. So what happens when we replace each of the coefficients with the absolute value of that coefficient? Well, you take the absolute value of a number, either it stays the same as it was before, or it flips from negative to positive. Now, "n" here, we know is at least one. So if any coefficient flips from negative to positive, then the overall number only goes up. So if we apply the absolute value of each of the coefficients we get an only bigger number. So T(n) is bounded above by the new polynomial where the coefficients are the absolute values of those that we had before. So why was that a useful step? Well now what we can do is we can play the same trick but with "n". So it's sort of annoying how right now we have these different powers of "n". It would be much nicer if we just had a common power of "n", so let's just replace all of these different "n"s by "n^k", the biggest power of "n" that shows up anywhere. So if you replace each of these lower powers of "n" with the higher power "n^k", that number only goes up. Now, the coefficients are all non negative so the overall number only goes up. So this is bounded above by "the absolute value of a<u>k" "n^k"</u> ...up to "absolute value of a<u>1" "n^k" ...plus "a<u>0" "n^k".</u></u> I'm using here that "n" is at least one, so higher powers of "n" are only bigger. And now you'll notice this, by our choice of "c" underlined in red, this is exactly equal to "c" times "n^k". And that's what we have to prove. We have to prove that T(n) is at most "c" times "n^k", given our choice of "c" for every "n" at least one. And we just proved that, so, end of proof. Now there remains the question of how did I know what the correct, what a workable value of "c" and "n<u>0"</u> were? And if you yourself want to prove that something is big O of something else, usually what you do is you reverse engineer constants that work. So you would go through a proof like this with a generic value of "c" and "n<u>0" and then</u> you'd say, "Ahh, well if only I choose "c" in this way, I can push the proof through." And that tells you what "c" you should use. If you look at the optional video on further examples of asymptotic notation, you'll see some examples where we derive the constants via this reverse engineering method. But now let's turn to a second example, or really I should say, a non-example. So what we're going to prove now is that something is not big O of something else. So I claim that for every "k" at least 1, "n^k" is not O(n^(k-1)). And again, this is something you would certainly hope would be true. If this was false, there'd be something wrong with our definition of big O notation and so really this is just to get further comfort with the definition, how to prove something is not big O of something else, and to verify that indeed you don't have any collapse of distinctive powers of ploynomials, which would be a bad thing. So how would we prove that something is not big O of something else? The most...frequently useful proof method is gonna be by contradiction. So, remember, proof by contradiction, you assume what you're trying to, establish is actually false, and, from that, you do a sequence of logical steps, culminating in something which is just patently false, which contradicts basic axioms of mathematics, or of arithmetic. So, suppose, in fact, n^k was big O of n^(k-1), so that's assuming the opposite of what we're trying to prove. What would that mean? Well, we just referred to the definition of Big O notation. If in fact "n^k" hypothetically were Big O of n^(k-1) then by definition there would be two constants, a winning strategy if you like, "c" and "n<u>0" such</u> that for all sufficiently large "n", we have a constant multiple "c" times "n^(k-1)" upper bounding "n^k". So from this, we need to derive something which is patently false that will complete the proof. And the way, the easiest way to do that is to cancel "n^(k-1)" from both sides of this inequality. And remember since "n" is at least one and "k" is at least one, it's legitimate to cancel this "n^(k-1)" from both sides. And when we do that we get the assertion that "n" is at most some constant "c" for all "n" at least "n<u>0". And this now</u> is a patently false statement. It is not the case that all positive integers are bounded above by a constant "c". In particular, "c+1", or the integer right above that, is not bigger than "c". So that provides the contradiction that shows that our original assumption that "n^k" is big O of "n^(k-1)" is false. And that proves the claim. "n^k" is not big O of "n^(k-1)", for every value of "k". So different powers of polynomials do not collapse. They really are distinct, with respect to big O notation.

# 2-4-Big Omega and Theta

0:00 In this lecture, we'll continue our formal treatment of asymptotic notation. We've already discussed big O notation, which is by far the most important and ubiquitous concept that's part of asymptotic notation, but, for completeness, I do want to tell you about a couple of close relatives of big O, namely omega and theta. If big O is analogous to less than or equal to, then omega and theta are analogous to greater than or equal to, and equal to, respectively. But let's treat them a little more precisely. The formal definition of omega notation closely mirrors that of big O notation. We say that one function, T of N, is big omega of another function, F of N, if eventually, that is for sufficiently large N, it's lower bounded by a constant multiple of F of N. And we quantify the ideas of a constant multiple and eventually in exactly the same way as before, namely via explicitly giving two constants, C and N naught, such that T of N is bounded below by C times F of N for all sufficiently large N. That is, for all N at least N naught. There's a picture just like there was for big O notation. Perhaps we have a function T of N which looks something like this green curve. And then we have another function F of N which is above T of N. But then when we multiply F of N by one half, we get something that, eventually, is always below T of N. So in this picture, this is an example where T of N is indeed big Omega of F of N. As far as what the constants are, well, the multiple that we use, C, is obviously just one half. That's what we're multiplying F of N by. And as before, N naught is the crossing point between the two functions. So, N naught is the point after which C times F of N always lies below T of N forevermore. So that's Big Omega. Theta notation is the equivalent of equals, and so it just means that the function is both Big O of F of N and Omega of F of N. An equivalent way to think about this is that, eventually, T of N is sandwiched between two different constant multiples of F of N. I'll write that down, and I'll leave it to you to verify that the two notions are equivalent. That is, one implies the other and vice versa. So what do I mean by T of N is eventually sandwiched between two multiples of F of N? Well, I just mean we choose two constants. A small one, C1, and a big constant, C2, and for all N at least N naught, T of N lies between those two constant multiples. One way that algorithm designers can be quite sloppy is by using O notation instead of theta notation. So that's a common convention and I will follow that convention often in this class. Let me give you an example. Suppose we have a subroutine, which does a linear scan through an array of length N. It looks at each entry in the array and does a constant amount of work with each entry. So the merge subroutine would be more or less an example of a subroutine of that type. So even though the running time of such an algorithm, a subroutine, is patently theta of N, it does constant work for each of N entries, so it's exactly theta of N, we'll often just say that it has running time O of N. We won't bother to make the stronger statement that it's theta of N. The reason we do that is because you know, as algorithm designers, what we really care about is upper bounds. We want guarantees on how long our algorithms are going to run, so naturally we focus on the upper bounds and not so much on the lower bound side. So don't get confused. Once in a while, there will a quantity which is obviously theta of F of N, and I'll just make the weaker statement that it's O of F of N. The next quiz is meant to check your understanding of these three concepts: Big O, Big Omega, and Big Theta notation. 3:29 So the final three responses are all correct, and I hope the high level intuition for why is fairly clear. T of N is definitely a quadratic function. We know that the linear term doesn't matter much as it grows, as N grows large. So since it has quadratic growth, then the third response should be correct. It's theta of N squared. And it is omega of N. So Omega of N is not a very good lower bound on the asymptotic rate of growth of T of N, but it is legitimate. Indeed, as a quadratic growing function, it grows at least as fast as a linear function. So it's Omega of N. For the same reason, big O of N cubed, it's not a very good upper bound, but it is a legitimate one, it is correct. The rate of growth of T of N is at most cubic. In fact, it's at most quadratic, but it is indeed, at most, cubic. Now if you wanted to prove these three statements formally, you would just exhibit the appropriate constants. So for proving that it's big Omega of N, you could take N naught equal to one, and C equal to one-half. For the final statement, again you could take N naught equal to one. And C equal to say four. And to prove that it's theta of N squared you could do something similar just using the two constants combined. So N naught would be one. You could take C1 to be one-half and C2 to be four. And I'll leave it to you to verify that the formal definitions of big omega, big theta, and big O would be satisfied with these choices of constants. One final piece of asymptotic notation, we're are not going to use this much, but you do see it from time to time so I wanted to mention it briefly. This is called little O notation, in contrast to big O notation. So while big O notation informally is a less than or equal to type relation, little O is a strictly less than relation. So intuitively it means that one function is growing strictly less quickly than another. So formally we say that a function T of N is little O of F of N, if and only if for all constants C, there is a constant N naught beyond which T of N is upper bounded by this constant multiple C times by F of N. So the difference between this definition and that of Big-O notation, is that, to prove that one function is big O of another, we only have to exhibit one measly constant C, such that C times F of N is upper bound, eventually, for T of N. By contrast, to prove that something is little O of another function, we have to prove something quite a bit stronger. We have to prove that, for every single constant C, no matter how small, for every C, there exists some large enough N naught beyond which T of N is bounded above by C times F of N. So, for those of you looking for a little more facility with little O notation, I'll leave it as an exercise to prove that, as you'd expect for all polynomial powers K, in fact, N to the K minus one is little O of N to the K. There is an analogous notion of little omega notation expressing that one function grows strictly quicker than another. But that one you don't see very often, and I'm not gonna say anything more about it. So let me conclude this video with a quote from an article, back from 1976, about my colleague Don Knuth, widely regarded as the grandfather of the formal analysis of algorithms. And it's rare that you can pinpoint why and where some kind of notation became universally adopted in the field. In the case of asymptotic notation, indeed, it's very clear where it came from. The notation was not invented by algorithm designers or computer scientists. It's been in use in number theory since the nineteenth century. But it was Don Knuth in '76 that proposed that this become the standard language for discussing rate of growth, and in particular, for the running time of algorithms. So in particular, he says in this article, "On the basis of the issues discussed here, I propose that members of SIGACT," this is the special interest group of the ACM, which is concerned with theoretical computer science, in particular the analysis of algorithms. So, "I propose that the members of SIGACT and editors in computer science and mathematics journals adopt the O, omega, and theta notations as defined above unless a better alternative can be found reasonably soon. So clearly a better alternative was not found and ever since that time this has been the standard way of discussing the rate of growth of running times of algorithms and that's what we'll be using here.

# 2-5-Additional Examples [Review - Optional]

0:00 This video is for those of you who want some additional practice with asymptotic notation. And we're gonna go through three additional optional examples. Let's start with the first one. So the point of this first example is to show how to formally prove that one function is big O of another. So the function that I want to work with is two raised to the N plus ten, okay, so it's the two to the N function that you're all familiar with, we're going to shift it by ten and the claim is that this function is big O of two to the N, so without the ten. So how would one prove such a claim? Well lets go back to the definition of what it means for one function to be big O over another, what we have to prove is we need to show that there exists two constants, so that for all sufficiently large N meaning N bigger than N-nought, our left hand side, so the function should be N plus ten is bounded above by a constant multiple C times the function on right hand side to the N. Right so for all sufficiently large N the function is bounded above by a constant multiple of two to the N. So unlike the first basic example where I just pulled the two constants out of a hat let's actually start the proof and see how you'd reverse engineer the suitable choice of these two constants. So, what a proof would look like, it would start with two to the N plus ten, on the left-hand side, and then there'd be a chain of inequalities, terminating in this, C times two to the N. So, let's just go ahead and start such a proof, and see what we might do. So, if we start with two to the N plus ten on the left-hand side, what would our first step look like? Well, this 10's really annoying, so it makes sense to separate it out. So you could write two to the N plus ten as the product of two terms. Two to the ten, and then the two to the N. Also known as just 1024 times two to the N. And now we're in, looking in really good shape. So if you look at where we are so far, and where we want to get to, it seems like we should be choosing our constant C to be 1024. So if we choose C to be 1024 and we don't have to be clever with N-nought we can just set that equal to one, then indeed star holds to the desired inequality and remember to prove that one function is big O of another all you gotta do is come up with one pair of constants that works and we've just reverse engineered it just choosing the constant C to be 1024 and N-nought to be one works so this proves that two to the N plus ten is big O over two to the N. Next let's turn to another non example how, of a function which is not big O over another. And so this will look superficially similar to the previous one. Instead of taking, adding ten in the exponent of the function two to the N, I'm gonna multiply by ten in the exponent. And the claim is if you multiply by ten in the exponent then this is not the same asymptotically as two to the N. So once again, usually the way you prove one thing is not big O of another is by contridiction. So we're going to assume the contrary, that two to the ten N is in fact big O of two to the N. What would it mean if that were true? Well, by the definition of big O notation, that would mean there are constant C and N-nought. So that for all large N, two to the ten N is bounded above by C times 2 to the N. So to complete the proof what we have to do is go from this assumption and derive something which is obviously false but that's easy to do just by cancelling this 2 of the N terms from both sides. So if we divide both sides by 2 to the N, which is a positive number since N is positive, what we find would be a logical consequence of our assumption would be that two raised to the nine N is bounded above by some fixed constant C for all N at least N-nought. But this inequality of course is certainly false. The right hand side is some fixed constant independent of N. The left hand side is going to infinity as N grows large. So there's no way this inequality holds for arbitrarily large N. So that concludes the proof by contradiction. This means our assumption was not the case, and indeed it is not the case that two to the ten N is big O of two to the N. So our third and final example is a little bit more complicated than the first two. It'll give us some practice using theta notation. Recall that while big O is analogous to saying one function is less than or equal to another, theta notation is in the spirit of saying one function is equal asymptotically to another. So here's gonna be the formal claim we're gonna prove, for every pair of functions F and G, both of these functions are defined on the positive integers, the claim is that it doesn't matter, up to a constant factors, whether we take point wise maximum of the two functions or whether we take the point wise sum of the two functions. So let me make sure it's clear that you know I mean by the point wise maximum by max F and G. So, if you look at the two functions, both functions of N, maybe we have F being this green function here and we have g hooked to this red function. Then by the point wise maximum max(F,G) just means the upper envelope of these two functions. So that's gonna be this blue function. So lets now turn to the proof of this claim that the point wise function of these two function is theta of the sum of two functions. So let's recall what theta means formally. What it means is that the function on the left can be sandwiched between the constant multiples of the function on the right. So we need to exhibit both the usual N-nought but also two constants, the small one, C1, and the big one, C2, so that the point wise maximum(F,G), whatever that may be, is wedged in between C1 and C2 times F(N) plus G(N), respectively. So to see where these constants C1 and C2 are going to come from, let's observe the following inequalities. So no matter what N is, any positive integer N, we have the following. Suppose we take the larger of F of N and G of N. And remember now, we've fixed the value of N, and it's just some integer, you know, like 23. And now F of N and G of N are theirselves, just numbers. You know, maybe they're 57 and 74, or whatever. And if you take the larger of F of N and G of N, that's certainly no more than the sum of F of N plus G of N. Now, I'm using, in this inequality, that F and G are positive. And that's something I've been assuming throughout the course so far. Here, I wanna be explicit about it, we're assuming that F and G cannot output negative numbers. Whatever integer you feed in, you get out something positive. Now, the functions we care about are things like the running time of algorithms, so there's really no reason for us to pollute our thinking with negative numbers. So, we're just gonna always be assuming in this class, positive numbers. And I'm actually using it here, the right hand side is the sum of two things, is bigger than just either one of the constituent summants. Secondly. If we double the larger of F of N and G of N well that's going to exceed the sum of F of N plus G of N, right? Because on the right hand side we have a big number plus a small number and then on the left hand side we have two copies of the big number so that is going to be something larger, now it's gonna be convenient it's gonna be more obvious what's going on if I divide both of these sides by two so that the maximum of F of N and G of N is at least half of F of N plus G of N that is at least half of the average and now we're pretty much home free right so what does this say. This says that for every possible N, the maximums wedged between suitable multiples of the sum. So one-half of F of N plus G of N. There's a lower bound on the maximum. This is just the second inequality that we derived. And by the first inequality that's bounded above by once times the sum. And this holds no matter what N is, [inaudible] at least one. And this is exactly what it means to prove that one function is theta of another. We've shown that for all N, not just for insuffiently large, but in fact for all N. The pointwise maximum of F and G is wedged between suitable constant multiples of their sum. And again, just to be explicit, the certifying choices of constants are N-nought equals one. The smaller constant is one half. And the bigger constant equals one. And that completes the proof.