wk2

# 0wk2-Overview

DIVIDE AND CONQUER ALGORITHMS: The next set of lectures discusses three non-trivial examples of the divide and conquer algorithm design paradigm. The first is for counting the number of inversions in an array. This problem is related to measuring similarity between two ranked lists, which in turn is relevant for making good recommendations to someone based on your knowledge of their and others' preferences ("collaborative filtering"). The second algorithm is Strassen's mind-blowing recursive algorithm for matrix multiplication, which improves over the obvious iterative method. The third algorithm, which is more advanced and is optional material, is for computing the closest pair of points in the plane.

THE MASTER METHOD: These lectures cover a "black-box" method for solving recurrences. You can then immediately determine the running time of most of the divide-and-conquer algorithms that you'll ever see! (Including Karatsuba's integer multiplication algorithm from Week 1.) The proof is a nice generalization of the recursion tree method that we used to analyze MergeSort. Ever wonder about the mysterious three cases of the Master Method? Watch these videos and hopefully all will become clear.

HOMEWORK: Problem Set #2 has five questions that should give you practice with divide-and-conquer algorithms and the Master Method. Programming assignment #2 asks you to implement the counting inversions algorithm (from Part III) in whatever programming language you please, run it on a quite large input, and enter the answer.

SUGGESTED READINGS FOR WEEK 2:

CLRS Chapter 4 (except Section 4.3), and Sections 28.1 and 33.4

DPV Sections 2.2 and 2.5

KT Sections 5.2-5.5

# 0wk2-menu

III. DIVIDE & CONQUER ALGORITHMS

Week 2 Overview10 分

O(n log n) Algorithm for Counting Inversions I12 分

O(n log n) Algorithm for Counting Inversions II16 分

Strassen's Subcubic Matrix Multiplication Algorithm22 分

O(n log n) Algorithm for Closest Pair I [Advanced - Optional]31 分

继续

O(n log n) Algorithm for Closest Pair II [Advanced - Optional]18 分

IV. THE MASTER METHOD

Motivation7 分

Formal Statement10 分

Examples13 分

Proof I9 分

Interpretation of the 3 Cases10 分

Proof II16 分

Problem Set #2

购买课程以解锁此项目。

测验: Problem Set #25 个试题

截止时间 六月 4, 11:59 晚上 PDT

Optional Theory Problems (Batch #1)10 分

Programming Assignment #2

购买课程以解锁此项目。

测验: Programming Assignment #21 个试题

截止时间 六月 4, 11:59 晚上 PDT

# 3-1-O(n log n) Algorithm for Counting Inversions I

In this next series of videos, we'll get some more practice applying the divide and conquer algorithm and design paradigm to various problems. This will also give us a glimpse of the kinds of application [inaudible] to which it's been successfully applied. We're gonna start by extending the merge sort algorithm to solve a problem involving counting the number of inversions of an array. Before we tackle the specific problem of counting the number of inversions in an array, let me say a few words about the divide and conquer paradigm in general. So again, you've already seen the totally canonical example of divide and conquer, namely merge sort. So the following three conceptual steps will be familiar to you. The first step, no prizes for guessing is you divide. The problem. Into smaller sub-problems. Sometimes this division happens only in your mind. It's really more of a conceptual step than part of your code. Sometimes you really do copy over parts of the input into say new arrays to pass on to your recursive calls. The second step, again no prizes here, is you conquer the sub-problems just using recursion. So for example, in Merge Sort, you conceptually divide the array into two different pieces. And then you [inaudible] with the conquer or sort to the first half of the array. And you, you do the same thing with the second half of the array. Now, of course, it's not quite as easy as just doing these two steps. Dividing the problem, and then solving the sub problems recursively. Usually, you have some extra cleanup work after the recursive calls, and to stitch together the solutions to the sub problems into one for the big problem, the problem that you actually care about. Recall, for example, in Merge Sort, after our recursive calls, the left half of the array was sorted, the right half of the array was sorted. But we still had to stitch those together. Merge into a sorted version of the entire array. So the [inaudible] step is to combine. The solutions to the subproblem into one problem. Generally the largest amount of ingenuity happens in the third step. How do you actually quickly combine solutions to subproblems into one to the original problem? Sometimes you also get some cleverness in the first step with division. Sometimes it's as simple as just spliting a ray in two. But there are cases where the division step also has some ingenuity. Now let's move on to the specific problem of counting inversions and see how to apply this divide and conquer paradygm. So let begin by defining the problem formally now. We're given as input an array A with a length N. And you can define the problem so that the array a contains any ole distinct numbers. But, let's just keep thing simple and assume that it contains the numbers one through n. The integers in that range in some order. That captures the essence of the problem. And the goal is to compute the number of inversions of this array so what's an inversion you may ask well an inversion is just a pair of array [inaudible] I and J with I smaller than J so that earlier array entry the I entry is bigger than the latter one the Jake one so one thing that should be evident is that if the array contains these numbers in sorted order if the array is simply one two three four all the way up to N then the number of inversions is zero. The converse you might also want to think through if the array has any other ordering of the numbers between one and N other than the assorted one, then it's going to have a non. Of zero number of inversions. Let's look at another example. So'spose we have an array of six entries. So the numbers one thru six in the following order. One, three, five followed by two, four, six. So how many inversions does this array have? So again what we need to look for are pairs of array entries so that the earlier or left entry is bigger than the later or right entry. So one example which we see right here would five and two. Those are right next to each other and out of order, the earlier entry is bigger than the other one. But there's others, there's three and two for example Those are out of order. And, five and four are also out of order. And I'll leave it to you to check that those are the only three pairs that are out of order. So summarizing the inversions in this array of length six are 3-2, 5-2, and 5-4. Corresponding to the array entries, 2-4, 3-4, and 3-5. Pictorially, we can think of it thusly, we can first. Write down the numbers in order, one up to six. And then we can write down the numbers again but, ordered in the way that their given in the input array. So, one three five two four six. And then we can connect the dots, meaning we connect one to one. Reconnect two to two, and so on. It turns out, and I'll leave to for you to, to think this through, that the number of crossing pairs of line segments prescisely correspond to the number of inversions. So we see that there are one, two, three crossing line segments. And these are exactly in correspondence with the three inversions, we found earlier. Five and two, three and two, and five and four. Now, [inaudible] wanna solve this problem you might ask. Well there's few reasons that come up. One would be to have a numerical similarity measure that quantifies how close to [inaudible] lists are to each other. So for example, suppose I took you and a friend, and I took, identified ten movies that both of you had seen. And I asked each of you to order, or to rank these movies from your most favorite to your least favorite. Now I can form an array, and compute inversions. And it quantifies, in some sense, how dissimilar your two rankings are to each other. So in more detail, in the first entry of this array, I would put down the ranking that your friend gave to your favorite movie. So if you had your favorite movie, Star Wars or whatever. And your friend only thought it was the fifth best out of the ten, then I would write down a five in the first entry of this array. Generally, I would take your second favorite movie. I would look at how your friend ranked that. I would put that in the second entry of the array and so on, all the way up to the tenth entry of the array, where I would put your friend's ranking of your least favorite movie. Now, if you have exactly identical preferences, if you rank them exactly the same way, the number of inversions of this array would be zero. And in general, the more inversions this array has, it quantifies that your lists look more and more different from each other. Now why might you want to do this why might you want to know whether two different people ranked things in the similar way had similar preferences well one reason might be what's called collaborative filtering, probably many of you have had the experience of going to a website and if you've made a few purchases through this website it starts recommending further purchases for you, so one way to solve this problem under the hood, is to look at your purchases look at what you seem to like, find other people who have similar preferences similar history look at things they've bought that you haven't, and then recommend. New products to you based on what similar customers seemed to have bought. So this problem captures some of the essence of identifying which customers or which people are similar based on data about what they prefer. So just to make sure we're all on the same page, let me pause for a brief quiz. We've already noticed that a given array will have zero inversions, if and only if it's in sorted order. If it only contains the numbers of one through N in order. So, on the other side, what is the largest number of inversions an array could possibly have? Let's say, just for an array of size six, like the one in this example here. So the answer to this question is the first one. Fifteen. Or in general in an N. Element array the largest number of inversions is N. Choose two. Also known as N times N minus one over two. Which, again, in the case of a [inaudible] is going to evaluate to fifteen. The reason is, the worst case is when the array is in backwards order, reverse [inaudible] order, and every single pair of [inaudible] indices is inverted. And so the number of indices IJ, with I less than J is precisely [inaudible] too. Let's now turn our attention to the problem of computing the number of inversions of an array as quickly as possible. So one option that is certainly available to us is the brute force algorithm. And by brute force I just mean we could set up a double four loop. One which goes through I, one which goes through J bigger than I, and we just check each pair IJ individually with I less than J whether that particular pair of array entities AI and AJ is inverted and if it is then we add it to our running count. And then we return the final count at the end of the double four loop. That's certainly correct. The only problem is, as we just observed, there's N [inaudible] two or a quadratic number of potential inversions so this algorithm's almost going to run in time quadratic in the array link. Now remember the mantra of any good algorithm designer. Can we do better? And the answer is yes. And the method we'll be using, divide and conquer. The way in which we'll divide will be motivated directly by merge sort where we recurs e separately on the left and the right half's of the array. We're gonna do the same thing here. To understand how much progress we can make purely using recursion let's classify the inversions of array into one of three types. So suppose we have an inversion of an array I, J, and remember in an inversion you always have I less than J. We're gonna call it a left inversion. If both of the array indices are at most N over two, where N is the array length. We're gonna call it a right inversion if they're both strictly greater than N over two. And we're gonna call it a split inversion if the smaller index is at most N over two and the larger index is bigger than N over two. We can discuss the progress made by recursion in these terms. When we recurse on the left-half of an array, if we implement our algorithm correctly, we'll successfully be able to count all of the inversions located purely in that first half. Those are precisely the left inversions. Similarly, a second recursive call on just the right half of an array, the second half of an [inaudible] array will successfully count all of the right inversions. There remains the questions of how to count the split inversions. But we shouldn't be surprised there's some residual work left over, even after the recursive calls do their job. That, of course, was the case at Merge Short, where [inaudible] magically took care of sorting the left half of the array, sorting the right half of the array. But there was still, after their return, the matter of merging those two sorted lists into one. And here again, after the recursion is gonna be the matter of cleaning up and counting the number of split inversions. So for example if you go back to the six element array we worked through before, 135246, you'll notice that there, in fact, all of the inversions are split. So the recursive calls will both come back counting zero inversions. And all of the work for that example will be done by the count split inversions subroutine. So let's summarize where things stand given underspecified high level description of the algorithm as we envision it. There is a base case. I'll go ahead and write it down for completeness, which is if we're given a one element array, then there's certainly no inversion so we can just immediately return the answer zero. For any bigger array, we're going to divde and conquer. So we'll count the left inversions with a recursive call. The right inversions with a recursive call. And then we'll have some currently unimplemented subroutine that counts the split inversions. Since every inversion is either left or right, or split, and can't be any more than one of those three, then, having done these three things, we can simply return their sum. So that's our high level attack on how we're gonna count up the number of inversions. And of course, we need to specify how we're gonna count the number of split inversions. And moreover, we lack that subroutine to run quickly. An analogy to emerge short, where, outside the recursive calls, we did merely linear work. Outs-, in the merge subroutine. Here, we'd like to do only linear work in counting up the number of split inversions. If we succeed in this goal, if we produce a correct and linear time of limitation to count up the number of split incursions, then this entire recursive algorithm will run in big O. Of N. Log in time. The reason the overall out rhythm will run in O. Of N. Log in time is exactly the same reason that merge short ran in N. Log in time. There's two recursive calls. Each on a problem of one-half the size. And outside of the recursive calls we would be doing linear work. So you could copy down exactly the same recursion tree argument we used for merge short. It would apply equally well here. Alternatively, very soon we will cover the master method, and as one very special case it will prove that this algorithm, if we can implement it thusly, will run in O. Of N. Log in time. Now one thing to realize, is this is a fairly ambitious goal, to count up the number of split inversions in linear time. It's not that there can't be too many split inversions. There can actually be a lot of them. If you have an array where the first half of the array contains the numbers N over two plus one, up to N. Whereas the second part of the array contains the numbers one up to N over two, that has a quadratic number of inversions, all of which are split. So, what we're attempting to do here is count up a quadratic number of things using only linear time. Can it really be done? Yes is can, as we'll see in the next video.

# 3-2-O(n log n) Algorithm for Counting Inversions II

So far we've developed a divide and conquer approach to count the number of inversions of an array. So we're going to split the array in two parts, recursively count inversions on the left, on the right. We've identified the key challenge is counting the number of split inversions quickly. Where a split inversion means that the earlier indexes on the left half of the array, the second index is on the right half of the array. These are precisely inversions that are going to be missed by both of our recursive calls. And the cracks or the problem is that there might be as many as quadratics but conversions. It somehow they go the run time they want. We need to do it in a linear time. So, here is the really nice idea which is going to let us do that. The idea is to piggy back on merge sort. By which I mean we're actually going to demand a bit more of our recursive calls to make the job of counting the number of split recursions easier. This is analogous to when you're doing a proof by induction, sometimes making the inductive hypothesis stronger, that's what lets you push through the inductive proof. So we're going to ask our recursive calls to not only count inversions in the array that they're passed, but also along the way to sort the array. And hey, why not? We know sorting is fast. Merge sort will do it in n log in time, which is the run time we're shooting for, so why not just throw that in? Maybe it'll help us in the combined step, and as we'll see, it will. So, what is this bias, why should we demand more recursive calls? Well, as we'll see in a couple of slides, the merge subroutine almost seem designed just to count the number of split inversions. As we'll see, as you merge two sorted sub arrays, you will naturally uncover, all of the split inversions. So, let me just be a little bit more clear about how our previous high level algorithm is going to now be souped up so that the recursive calls sort, as well. So, here is the high level algorithm we proposed before where we just recursively counted versions on the left side, on the right side. And then, we have some currently unimplemented subroutine counts splint if which is responsible for counting the number of split inversions. So we're just going to augment this as follows so instead of being called count now we're going to call it sort and count. That's going to be the name of our algorithm. The recursive calls, again, just invoke sort and count. And so now we know each of those will not only count the number of inversions in sub array, but also return a sorted version. So, out from the first one we're going to get arrayed B back which is the sorted version of the array that we past it and we got the sorted array C back from the second recursive call or sort of version of the array that we past it. And now, the counts split inversions now, in addition to count and split inversions is responsible for merging the two sort of subarrays, B and C. So CountSplitInv will be responsible for outputting an array D, which is a sorted version of the original input array A. And so I should also rename our unimplemented subroutine to reflect its now more ambitious agenda. So we'll call this mergeAndCountSplitInv. Now, we shouldn't be intimidated by asking our combining subroutine to merge the two sorted subarrays B and C, because we've already seen, we know how to do that in linear time. So the question is just, piggybacking on that worth, can we also count the number of split inversions in an additional linear time? We'll see that we can, although that's certainly not obvious. So you should again at this point have the question why are we doing this? Why are we just making ourselves do more work? And again the hope is that the payoff is somehow counting split inversions becomes easier by asking our recursive calls to give some additional work of sorting. So to develop some intuition for why that's true. Why merging naturally uncovers the number of splits inversions. Let's recall with just the definition of the original merge subroutine from merge sort was. So here's the same pseudocode we went through several videos ago. I have renamed the letters of the arrays to be consistent with the current notation. So we're given two sorted subarrays. These come back from a recursive calls. I'm calling them B and C. They both have length n/2 and were responsible for producing the sorted combination of B and C so that's an output array D of length n. And again the ideas simple, you just take the two sorted sub-arrays B and C and then you take the output array D which you're responsible for populating. And using an index k you're going to traverse the output D from left to right. That's what this outer form here does and you're going to maintain pointers i and j to the sorted sub arrays B and C respectively. And, the only observation is that whatever the minimum element that you haven't copied over to D yet is, it's got to be either the left most element of B that you haven't seen yet or the left most element of C that you haven't seen yet. B and C by virtue of being sorted, the minimum element remaining has to be the next one available to either B or C. So you just proceed in the obvious way. You compare the two candidates for the next ones that copy over. You look at B(i). You look at C(j). Whichever one is smaller, you copy over, so the first part of the if statement is for when B contains the smaller one. The second part of the else statement is for when C contains the smaller one, okay? So, that's how merge works. You go down B and C in parallel, populating D in sorted order from left to right. Now to get some feel for what on Earth any of this has to do with the split inversions of an array, I want you to think about an input array A that has the following property. That has the property that there are no split inversions whatsoever. So every inversion in this input array A is going to be either a left inversion, so both indices are at most n/2, or a right end version. So both indexes are strictly greater than n/2. Now, the question is, given such an array A, once you're merging at this step, what do the assorted subarrays B and C look like for an input array that has no split inversions? The correct answer is the second one. That if you have an array with no split inversions then everything in the first half is less than everything in the second half, why? Well, consider the contra-positive. Suppose you had even one element in the first half which was bigger than any element in the second half, that pair of elements alone would constitute a split inversion, okay? So if you have no split inversions then everything on the left is smaller than everything in the right half of the array. Now, more to the point, think about the execution of the merge subroutine on an array with this property, on an input array A where everything in the left half is less than everything in the right half. What is merge going to do? All right, just remember it's always looking for whichever is smaller the first element of remaining in B or the first element remaining in C and that's what it copies over. When everything in B is less than everything in C everything in B is going to get copied over in to the output array D before C ever gets touched. Okay, so merge has an unusually trivial execution on input arrays with no split inversions with zero split inversions First it just goes through B and copies it over then it just concatinate C. Okay, there's no interweaving between the two. So, no split in versions means nothing it copied from C, until it absolutely has to, until B is exhausted. So, this suggests that, perhaps, copying elements over from the second sub-array C has something to do with the number of split inversions in the original array, and that is in fact the case. So we're going to see a general pattern about copies from the second array C through the output array, exposing split inversions in the original input array A. So let's look at a more detailed example to see what that pattern is. So let's return to the example in the previous video, which is an array with six elements, ordered 1, 3, 5, 2, 4, 6. So we do our recursive call and in fact, the left half of the array is sorted and the right half of the array is already sorted. No sorting was going to be done and I'm actually going to get zero inversions for both our recursive calls. Remember in this example it turns out all of the inversions are split versions. So now let's trace through the merge sub routine invoked on these two sorted subarrays. And try to spot a connection with the number of split inversions in the original six element array. So we initialize indices i and j to point to the first element of each of these subarrays. So this left one is B and this right one is C and the output is D. And the first thing we do is we copy the 1 over from B into the top of array so 1 goes there and we advance this index over to the 3. And here, nothing really interesting happens, there's no reason to count on this split inversions and indeed the number one is not involved at any split inversions, because you want it smaller than all of the other elements and it's also in the first index. Things are much more interesting when we copy over the element 2 from the second array C. And notice, at this point, we have diverged from the trivial execution that we would see with an array with no split inversions. Now we're copying over something from C before we've exhausted copying B. So we are hoping this will expose some split inversions. So we copy over the two and we advance the second pointer j into C and the thing to notice is, this exposes two split inversions. The two split inversions that involve the element two. And those inversions are 3,2 and 5,2. So why did this happen? Well the reason we copied two over is because it's smaller than all the elements we haven't yet looked at in both B and C. So in particular 2 is smaller than the remaining elements in B, the 3 and the 5. But also because B is the left array, the indices of the 3 and the 5 have to be less than the index of this 2. So, these are inversions, 2 is further to the right in the original input array, and yet it's smaller than these remaining elements in B. So, there are two elements remaining in B, and those are the two split versions that involve the elements two. So, now let's go back to the merging subroutines, and what happens next. Well, next we'll make a copy from the first array and we sort of realize that nothing really interesting happens when we copy it from the first array, at least with respect to split in versions. Then we copy the four over, and yet again, we discover a split inversion, the remaining one, which is 5,4. Again, the reason is, given that 4 was copied over before what's left in B, it's got to be smaller than it, but by virtue of being in the rightmost array, it's also not going to have a bigger index, so it's gotta be a split inversion. Now the rest of the merge subroutines executes without any real incident. Five gets copied over and we know copies from the left array are boring and then we copy the six over and copies from the right array are generally interesting but not if the left array is empty. That doesn't involve any split versions. And you will recall from the earlier video that these were the inversions in your original array, 3252 and 54. We discovered them all on an automated method by just keeping an eye out when we copy from the right array C. So this is indeed a general principle so let me state the general claim. So, the claim is not just in this specific example, in this specific execution. But no matter what the inquiry is, no matter how many split inversion there might be, the split inversions that involve an element of the second half of the array are precisely those elements remaining in the first array when that element gets copied over to the output array. So this is exactly the pattern that we saw in the example. What were, so on the right array C, we have the elements two, four and six. Remember every split version has to, by definition, involve one element from the first half and one element from the second half. So the count for split inversions, we can just group them according to which element of the second array that they involve. So out of the two four and six, the two is involved in the split up inversions three two and five two. The three and the five were exactly the elements remaining in B when we copied over two. The split inversions involving four is exactly the inversion five, four and five is exactly the element that was remaining. In B when we copied over the four, there's no split inversions involving six and indeed, the element B was empty when we copied the six over in the output array D. So what's the general argument? Well it's quite simple. Let's just zoom in and fixate on a particular element x that belongs to that first half of that array. That's amongst the first half of the element. And let's just examine which y's, so which elements of the second array, the second half of the original input array, are involved in split inversions with x. So there are two cases, depending on whether x is copied over into the output array D before or after y. Now if x is copied to the output before y, well then since the output's in sorted order it means x has got to be less than y so there's not going to be any split inversion. On the other hand if y is copied to the output d before x then again because we populate the left to right in sorted order, that's got to mean that y is less than x. Now x is still hanging out in the left array so it has a less index than y, y comes from the right array so it's not a split inversion. So putting these two together, it says that the elements x of the array B that form split inversions with y are precisely those that are going to get copied to the output array after y. So those are exactly the number of elements remaining in B when y gets copied over. So that proves the general claim. So this slide was really the key insight. Now that we understand exactly why counting split inversions is easy, as we're merging together two sorted subarrays, it's a simple matter to just translate this into code and get a linear time of notation of a sub routine that both emerges and counts the number of split inversions. Which then in the overall course of the algorithm we'll have n log n running time just as in merge sort. So, let's just spend a quick minute filling in those details. So, I'm not going to write up the pseudo code. I'm just going to write what you need to augment the merge pseudo code discussed a few slides ago by in order to count split inversion as you're doing the merging. And this will follow immediately from the previous plan which indicated how split version relate to the number of elements remaining in the left array as you're doing the merge. So the idea is the natural one, as you're doing the merging, according to the previous pseudo code, of the two sorted subarrays you just keep a running total of the number of split inversions that you encounter. And so you've got your sorted subarray B, you've got your sorted subarray C. You're merging these into an output array D, and as you traverse through D and k goes from 1 to n, you just start out at zero and increment it by something each time you copy over from either B or C. So, what's the increment? Well, what did we just see, we saw the copies involving B don't count, we're not going to look at split inversions when to copy over from B, only when we look at them from C, right? Every split inversion involves exactly one element from each of B and C. So, I may as well count them via the elements in C and how many split inversions are involved with the given element of C, well it's exactly how many elements of B remain when it gets copied over. So, that tells us how to increment this running count. And, it follows immediately from the claim on the previous slide that this implementation of this running total counts precisely the number of split inversions that the original input array A possesses. And we'll call that the left inversions are counted by the first recursive call of the right inversions are counted by the second recursive call. Every inversion is either at left or right or splitt that's exactly one of those three types. So, with our three different subroutines, the two recursive ones and this one here, we successfully count of all the inversions of the original input array. So that's the correctness of the algorithm. What's the running time? We'll recall in mergesort, we began just by analyzing the running time of merge and then we discussed the running time of the entire mergesort algorithm. Let's do the same thing here briefly. So what's the running time of the subroutine for this merging and simultaneously counting the number of split inversions? Well there's the work that we do in the merging, and we already know that that's linear. And then the only additional work here is incrementing this running count, and that's constant time for each element of D, right? Each time we do a copy over we do some single addition to our running count. So constant time for element of D, or linear time over all. So, I'm being a little sloppy here. Sloppy in a very conventional way but it is a little sloppy by writing O(n) + O(n) = O(n). Be careful when you make statements like that. Right, so, if you added O(n) to itself n times, it would not be O(n), but if you add O(n) to itself a constant number of times, it is still O(n). So you might, as an exercise, want to write out a formal version of what this means. Basically there's some constant c1 so that the merge steps takes at most c1 n steps. There's a constant c2 so that the rest of the work is at most c2 times n steps. So when you add them, we get it's at most quantity c1 plus c2 times n steps, which is still big O(n), because c1 plus c2 Is a constant, okay? So, linear work for merge, linear work for the running count, so does linear work in the subroutine overall. And now, by exactly the same argument, we'll use in merge sort because we have two reversing calls in half the size. And with your linear work outside the recursive calls, the overall running time is O(n) log n. So, it really just piggybacked on merge sort upped to the constant factor a little bit to do the counting along the way, but the running time remains the big O(n log n).

# 3-3-Strassen's Subcubic Matrix Multiplication Algorithm

In this video, we'll apply the divide and conquer algorithm design paradigm to the problem of multiplying matrices. This will culminate in the study of Strassen matrix multiplication algorithm. And this is a super cool algorithm for two reasons. First of all, Strassen's algorithm is completely non-trivial. It's totally non-obvious, very clever, not at all clear how Strassen ever came up with it. The second cool feature is it's for such a fundamental problem. So computers as long as they've been in use from the time they were invented up til today a lot of their cycles is spent multiplying matrices. It just comes up all the time in important applications. So let me first just make sure we're all clear on what the problem is of multiplying two matrices. So we're going to be interested in three matrices x, y and z. I'm going to assume they all have the same dimensions, n x n. The ideas we'll talk about are also relevant for multiplying non square matrices but we're not going to discuss it in this video. The entries in these matrices, you could think of it as whatever you want. Maybe they're integers, maybe they're rationals. Maybe they're from some finite field. It depends on the application but the point is they're entries that we can add and multiply. So how is it that you take two n x n matrices x and y and multiply them producing a new n x n matrix z? Well recall that the ij entry of z, that means the entry in the ith row and the jth column, is simply the dot product of the ith row of x with the jth column of y. So if ij was this red square, this red entry over in the z matrix that would be derived from the corresponding row of the x matrix and the corresponding column of the y matrix. And recall what I mean by dot product, that just means you take the products of the individual components and then add up the results. So ultimately the zij entry boils down to a sum over n things where each of the constituent products is just the xik entry, the ik entry of the matrix x with a kj entry of the matrix y. Where your k is ranging from 1 to n. So that's how zij is defined for a given pair of indices i and j. One thing to note is that where we often use n to denote the input size, here we're using n to denote the dimension of each of these matrices. The input size is not n, the input size is quite a bit bigger than n. Specifically, each of these are n x n matrices that contains n squared entries. So since presumably we have to read the input, which has size n squared, and we have to produce the output, which also has size n squared. The best we could really hope for a matrix multiplication algorithm would be a running time of n squared, so the question is how close can we get to it. Before we talk about algorithms for matrix multiplication let me just make sure we're all crystal clear on exactly what the problem is so let's just actually spell out what would be the result of multiplying two different 2 x 2 matrices. So we can parameterize two generic 2 x 2 matrices by just giving the first one entries a, b, c, and d. Or these four entries could all be anything. And then, we're multiplying by a second 2 x 2 matrix. Let's call its entries e, f, g, and h. Now what's the result of multiplying these? Where again, it's going to be a 2 x 2 matrix where each entry is just the corresponding dot product of the relevant row of the first matrix and column of the second matrix. So to get the upper left entry we take the dot product of the upper row of the first matrix and the first column of the left column of the second matrix so that results in ae + bg. To get the upper right entry we take the dot product of the top row of the left matrix with the right column of the second matrix, so that gives us af + bh and then filling in the other entries in the same way, we get ce + dg and cf + dh. Okay, so that's multiplying two matrices, and we've already discussed the definition in general. Now, suppose you had to write a program to actually compute the result of multiplying two n x n matrices. One natural way to do that would just be to return to the definition, which defines each of the n squared entries in the z matrix as a suitable sum of products of entries of the x and y matrices. So in the next quiz I'd like you to figure out exactly what would be the running time of that algorithm as a function of the matrix dimension n. Where, as usual, we count the additional multiplication of two individual entries as a constant time operation. So the correct response to this quiz is the third answer. That the running time of the straightforward iterative algorithm runs in cubic time relative to the matrix dimension n. To see this just recall what the definition of the matrix multiplication was. The definition tells us that each entry zij of the output matrix z is defined as the sum from k = 1 to n of xik times ykj. That is the dot product of the ith row of the x matrix and the jth column of the y matrix. I'm certainly assuming that we have the matrices represented in a way that we could access a given entry in a constant time. And under that assumption remember each of these products only takes constant time. And so then to compute zij we just have to add up these n products so that's going to be theta of n time to compute a given zij and then there's an n squared entries that we have to compute. There's n choices for i, n choices for j. So that gives us n squared times n or cubic running time overall. For the natural algorithm, which is really just a triple for loop which computes each entry of the output array separately using the dot product. So the question as always for the keen algorithm designer is, can we do better. Can we beat n cube time for multiplying two matrices. And we might be especially emboldened with the progress that we've already seen in terms of multiplying two integers. We apply the divide and conquer algorithm to the problem of multiplying two integers. And we had both a naive recursive algorithm and a seemingly better algorithm due to Gauss, which made only three recursive calls. Now we haven't yet analyzed the running time of that algorithm. But as we'll see later, that does indeed beat the quadratic running time of the grade school algorithm. So it's very natural to ask, can we do exactly the same thing here? There's the obvious n cubed algorithm which follows straight from the definition, perhaps analogous to Gauss we could have some clever divide and conquer method which beats cubic time. So that's what we're going to explore next. Let's recall the divide and conquer paradigm, what does it mean to use it? Well we first have to identify smaller subproblems, so if we want to multiple two n x n matrices. We have to identify multiplications of smaller matrices that we can solve recursively. Once we figured out how we want to divide the given problem into smaller ones. Then the conquer step, we simply invoke our own algorithm recursively. That's going to recursively multiply the smaller matrices together. And then in general, we'll have to combine the results of the recursive cause to get the solution for the original problem. In our case, to get the product of the original two matrices from the product of whatever submatrices we identify. So how would we apply the divide and conquer paradigm to matrices? So we're given two n x n matrices, and we have to somehow identify smaller pairs of square matrices that we can multiply recursively. So the idea I think is fairly natural. So we start with a big n by n matrix x, right, so there's n rows and n columns. And we have to somehow divide it into smaller pieces. Now the first thing you might think about is you put it into it's left half and it's right half analogous to what we've been doing. With arrays, but then we're going to break X into two matrices which are no longer square, which are n over 2 in one dimension, and have length n in the other dimension. And we want to recursively call a subroutine that multiplies square matrices. So what seems like the clear thing to do, is to divide X into quadrants. Okay, so we have four pieces of X, each is going to be n over 2 by n over 2 corresponding to the different quarters of this matrix. So let's call these different quadrants or blocks in matrix terminology A, B, C, and D. All of these are n over 2 by n over 2 matrices. As usual for simplicity, I'm assuming that n is even. And as usual it doesn't really matter and we can do the same trick with Y. So, we'll divide Y into quadrants. N over 2 by n over 2 matrices, which we'll call E, F, G, and H. So one thing that's cool about matrices is when you split them into blocks and you multiply them, the blocks just behave as if they were atomic elements. So what I mean by that, is that the product of X and Y can be expressed in terms of its quadrants. And each of its four quadrants, each of its four corners can be written as a suitable arithmetic expression of the quadrants of X and Y. So here's exactly what those formulas are. They're exactly analogous to when we just multiplied a pair of 2 by 2 matrices. So I'm not going to formally prove this fact. I'm sure many of you have seen it before or familiar with it. And if you haven't, it's actually quite easy to prove. It's not obvious since you can't see it off the top of your head necessarily. But if you go back to the definition, it's quite easy to verify. But indeed when you multiply X and Y, you can express its quadrants into blocks in terms of the blocks of X and Y. So what we just did is completely analogous to when talking about integer multiplication, and we wanted to multiply two integers, x and y, and we broke them into pairs of n over 2 digits. And then we just took the expansion, and we've observed how that expansion could be written in terms of products of n over 2 digit numbers. It's the same thing going on here, except with matrices. So now we're in business as far as a recursive approach. We want to multiply x and y. They're n by n matrices. We recognize, we going to express that product, x times y. In terms of the products of n over 2 by n over 2 matrices. Things were able to multiply recursively, plus additions. And additions, it's clearly easy to multiply two different matrices with, say, n squared entries each. It's going to be linear in the number of entries. So it's going to be n squared time to add two matrices that are n by n. So this immediately leads us to our first recursive algorithm. To describe it, let me quickly rewrite that expression we just saw on the previous slide. And now our first recursive algorithm is simply to evaluate all of these expressions in the obvious way. So specifically in step one, we recursively compute all of the necessary products. And observe that there are eight products that we have to compute. Eight products with n over 2 by n over 2 matrices. There are four entries in this expansion of x times y. Each of the blocks is the sum of two products and none of the products reoccurred. They're all distinct. So naively, if you want to evaluate this, we have to do eight different products of n over 2 by n over 2 matrices. Once those recursive calls complete, then all we do is do the necessary four additions. As we discussed, so that takes time proportional to the number of entries in the matrix. So this is going to take a quadratic time overall, quadratic in n. Linear in the number of entries. Now the question you should be asking is, is this a good algorithm? Was this good for anything? This recursive approach. Splitting x and y into these blocks. Expanding the product in terms of these blocks then recursively computing each of the blocks. And I want to say it's totally not obvious. It is not clear what the running time of this recursive algorithm is. I'm going to go ahead and give you a spoiler which is going to follow from the master method that we'll talk about in the next lecture. But it turns out, with this kind of recursive algorithm where you do eight recursive calls. Each on a problem with dimension half as much as what you started with, and then do quadratic time outside, the running time is going to be cubic. So exactly the same as with the straightforward iterative algorithm that follows from the definition. That was cubic, it turns out, and that was clearly cubic. This one, although it's not obvious, is cubic as well. So no better, no worse than the straightforward iterative algorithm. So in case you're feeling disappointed that we went through all this work in this sort of seemingly clever divide and conquer approach for matrix multiplication and came out at the end no better than the iterative algorithm. Well, there's really no reason to despair. because remember back in integer multiplication, we had a straightforward recursive algorithm. Where we have to do four recursive calls. Products of n over 2 digit numbers. But then we had Gauss' trick, which said if we only did more clever products and more clever additions and subtractions, then we can get away with only three recursive calls. And we'll see later if that is indeed a significant savings in the time we did for integer multiplication. And we've done nothing analogously clever to Gauss' trick for this matrix multiplication problem. All we did is the naive expansion, in terms of submatrices, and the naive evaluation of the resulting expressions. So, the $64,000 question is then, can we do something clever, to reduce the number of recursive calls, from 8 down to something lower? And that is where Strassen's Algorithm comes in. So the high level Strassen's Algorithm has two steps, just like the first recursive algorithm that we discussed. It recursively computes some products of smaller matrices and over to a roughly n over 2 by n over 2 matrices. But there's only going to be seven of them. But they will be much less straightforward, they will be much more cleverly chosen than in the first recursive algorithm. In step two then is to actually produce the product of x and y. Produce each of those four blocks that we saw. With suitable additions and subtractions of these seven products. And again, these are much less straightforward than in the first recursive algorithm. And so, while the additions and subtractions involved will be a little bit more numerous than they were in the naive recursive algorithm. It's only going to change the work in that part of the algorithm by a constant factor. So we'll still spend only theta (n squared) work adding and subtracting things, and we get a huge win in decreasing the number of recursive calls from 8 to 7. Now just in case you have the intuition that shaving off one of eight recursive calls should only decrease the running time of an algorithm by one-eighth by 12.5%. In fact it has a tremendously more amplified effect. Because we do one less recursive call over and over and over again as we keep recursing in the algorithm. So it makes a fundamental difference in the eventual running time of the algorithm as we'll explore in detail in the next set of lectures when we discuss the master method. So again a bit of a spoiler alert. What you're going to see in the next set of lectures that indeed Strassen's Algorithm does beat cubic time. It's better than n cubed time. You'll have to watch the next set of lectures if you want to know just what the running time is. But I'm going to tell you now that savings of the eighth recursive call is what changes the algorithm from cubic to subcubic. Now, 1969 was, obviously, quite a bit before my time. But by all accounts, from people I've talked to who were around then and from what the books say, Strassen's Algorithm totally blew people's minds at the time. Everybody was just assuming that there's no way you could do better than the iterative algorithm, the cubic algorithm. It just seemed that matrix multiplication intuitively fundamentally required all of the calculations that are spelled out in the definition. So Strassen's Algorithm is an early glimpse of the magic and of the power of clever algorithm design. But if you really have a serious ingenuity, even for super fundamental problems, you can come up with fundamental savings over the more straightforward solution. Solutions. So those are the main points I wanted to talk about Strassen's algorithm. How you can beat cubic time by saving a recursive call with suitably chosen clever products and clever additions and subtractions. Maybe a few of you are wondering what are these cleverly chosen products, can you really do this? And I don't blame you. There's no reason to believe me just because I sort of spelled out this high level idea. It's not obvious this should work. You might want to actually see the products. So for those of you like that, this last slide is for you. So here is Strassen's Algorithm in it's somewhat gory detail. So let me tell you what the seven products are that we're going to form. I'm going to label them P1 through P7 and they're all going to be defined in terms of the blocks of the input matrices x and y. So let me just remind you that we think of x in terms of its blocks A, B, C, D and we think of y in terms its blocks E, F, G, H. And remember a through h are all n over 2 by n over 2 sub-matrices. So here are the seven products P1 though P7. P1 = A(F-H), P2 = (A+B)H, P3 = (C+D)E. P4 = D(G-E). P5 = (A+D)(E+H). P6 = (B-D)(G+H) and finally P7 = (A-C)(E+F). So I hope you'll agree that these are indeed only seven products. And we could compute these with seven recursive calls. So we've pre-processed with a little bit of additions and subtractions. We have to compute F minus H, A plus B, C plus D, and so on. We compute all these new matrices from the blocks. And then we can recursively, with seven recursive calls. Do the seven products that operate on n over 2 by n over 2 matrices. Now the question is why is this useful, why on earth would we want to know the values of these seven products. So the amazing other part of the algorithm is that from just these seven products we can using only addition and subtraction recover all four of the blocks of x times y. So x times y you recall we solved it in terms of blocks. So we previously computed this to be AE+BG in the upper left corner and similarly expressions for the upper right, lower left or lower right blocks. So this we already know. So the content of the claim is that these four blocks also arise from the seven products in the following way. So the claim here is that these two different expressions for x times y are exactly the same, and they're the same block by block. So in the other words, what the claim is that this crazy expression P5+P4-P2+P6 where those are four of the products that we have listed above, that is precisely AE+BG. Similarly we're claiming the P1+P2 as exactly AF+BH that's actually easy to see P3+P4 is CE+DG. That's also easy to see. And then the other complicated one is the P1+P5-P3-P7 is exactly the same as CF+DH, so all four of those hold. So, let me, just so you believe me because I don't know why you'd believe me unless I showed you some this derivation. Let's just look at the proof of one of the cases of the upper left corner. So that is let's just expand out this crazy expression, P5+P4-P2+P6. What do we get? Well, from P5 we get, AE+AH+DE+DH, then we add P4, so that's going to give us, +DG-DE, then we subtract P2, so that gives us a -AH-BH, and then we add in P6, so that gives us a BG+BH-DG-DH. Okay so what happens next, well now we look for cancellations, so we cancel the AH's we cancel the DE's, we cancel the DH's, we cancel the DG's, we cancel the BH's and holy cow, what do we get? We get AE+BG, that is we get exactly what we were supposed to in the upper left block of x times y. So we just actually verified that this equation holds for the upper left block. It's quite easy to see that it holds for the upper right and lower left blocks. And then a comparable calculation verifies it for the lower right blocks of the two. So summarizing, because this claim holds, because actually we can recover the four blocks of x times y from these seven products. Strassen's algorithm works in the following way. You compute the seven products P1 through P7 using seven recursive calls. Then you just compute the four blocks using some extra additions and subtractions as shown in the claim. So it's seven recursive calls on n over 2 by n over 2 matrices. Plus n squared work due to the necessary additions and as you'll see in the master method lecture that is actually sufficient for subcubic time. Now, I sympathize with you if you have the following question, which is how on earth did Strassen come up with this. And indeed this sort of illustrates, the difference between checking somebody's proof and coming up with a proof. Given that I told you the magical seven products and how you from them you can recover the four desired blocks of x times y. It's really just mechanical to see that it works. It's a totally different story of how do you come up with P1 through P7 in the first place. So how did Strassen come up with them? Honestly, your guess is as good as mine.

# 3-4-O(n log n) Algorithm for Closest Pair I [Advanced - Optional]

So in this video and the next, we're going to study a very cool divide and conquer algorithm for the closest pair problem. this is a problem where you're given n points in the plane and you want to figure out which pair of points are closest to each other. So this would be the first taste we get of an application in computational geometry, which is the part of algorithms which studies how to reason and manipulate geometric objects. So those algorithms are important in, among other areas robotics, computer vision and computer graphics. So this is relatively advanced material, it's a bit more difficult than the other applications of divide and conquer that we've seen. The algorithm's a little bit tricky and it has a quite nontrivial proof of correctness, so just be ready for that and also be warned that because it's more advanced I'm going to talk about the material in at a slightly faster pace tha I do in most of the other videos. So let's begin now by defining the problem formally, so we're given as imput endpoints in the plane, so each one just define by its x coordinate and ist y coordinate. And when we talk about the distance between two points in this problem, we're going to focus on Euclidean distance. So, let me remind you what that is briefly, but we're going to introduce some simple notation for that, which we'll use for the rest of the lecture. So we're just going to note the Euclidean distance between two points, pi and pj, by d of pi pj. So in terms of the x and y coordinates of these two points, we just look at the squared differences in each coordinate, sum them up, and take the square root. And now as the name of the problem would suggest, the goal is to identify among all pairs of points that pair which has the smallest distance between them. Next, let's start getting a feel for the problem by making some preliminary observations. First, I want to make an assumption purely for convenience that there's no ties. So that is I'm going to assume all endpoints have distinct x coordinat es, and also all endpoints have distinct y coordinates. It's not difficult to extend the algorithm to accommodate ties. I'll leave it to you to think about how to do that. So next, let's draw some parallels with the problem of counting inversions, which was a earlier application of divide and conquer that we saw. The first parallel I want, want to out is that, if we're comfortable with the quadratic time algorithm, then this is not a hard problem, we can simply solve this by brute-force search. And again, by brute-force search, I just mean we set up a double for loop, which iterates over all distinct pairs of points. We compute the distance for each such pair and we remember the smallest. That's clearly a correct algorithm, it has to iterate over a quadratic number of pairs, so its running time is going to be theta of n squared. And, as always, the question is can we apply some algorithmic ingenuity to do better? Can we have a better algorithm than this naive one which iterates over all pairs of points? You might have a, an initial instinct that because the problem asks about a quadratic number of different objects, perhaps we fundamentally need to do quadratic work. But again, recall back in counting inversions, using divide and conquer, we were able to get an n log n algorithm despite the fact that there might be as many as a quadratic number of inversions in an array. So the question is, can we do something similar here for the closest pair problem? Now, one of the keys to getting an n log n time algorithm for counting inversions was to leverage a sorting subroutine. Recall that we piggybacked on merge sort to count the number of inversions in n log n time. So the question is, here, with the closest pair problem, perhaps, sorting again can be useful in some way to beat the quadratic barrier. So, to develop some evidence that sorting will indeed help us compute the closest pair of points embedded in quadratic time, let's look at a special case of the problem, really, an easier version of t he problem, which is when the points are just in one dimension, so on the line rather that in two dimensions in the plane. So in the 1D version, all the points just lie on a line like this one, and we're given the points in some arbitrary order not necessarily in sorted order. So, a way to solve the closest pair problem in one dimension, is to simply sort the points, and then of course, the closest pair better be adjacent in this ordering, so you just iterate through the n minus 1 consecutive pairs and see which one is closest to each other So, more formally, here's how you solve the one-dimensional version of the problem. You sort the points according to their only coordinate, because you're going to remember, this is one dimension. So as we've seen, using merge sort, we can sort the points in n log n time and then we just do a scan through the points, so this takes linear time. And for each consecutive pair, we compute their distance and we remember the smallest of those consecutive pairs and we return that. That's gotta be the closest pair. So, in this picture here on the right, I'm just going to circle here in green the closest pair of points. So this is something we discover by sorting and then doing a linear scan. Now, needless to say, this isn't directly useful, this is not the problem I started out with. We wanted to find out the closest pair among of points in the plane not points in the line. But, I want to point out that, this, even in the line, there are a quadratic number of different pairs, so brute-force search is still a quadratic time algorythm even in the 1D case. So at least, with one dimension, we can use sorting, piggyback on it, to beat the naive brute-force search bound and solve the problem in n log n time. So our goal for this lecture is going to be to devise an equally good algorithm for the two-dimensional case, so we want to solve closest pair of points in the plane, again, in n log n, n time. So we will succeed in this goal. I'm going to show you an n log n time algo rithm for 2D closest pair. It's going to take us a couple steps. So let me begin with a high level approach. Alright. So the first I need to try is just to copy what worked for us in the one-dimensional case. So the one-dimensional case, we first sorted the points by their coordinate and that was really useful. Now, in the 2D case, points have two coordinates, x coordinates and y coordinates, so there's two ways to sort them. So let's just sort them both ways, that is, the first step of our algorithm, which you should really think of as a preprocessing step. We're going to take the input points. We invoke merge sort once to sort them according to x coordinate, that's one copy of the points. And then we make a second copy of the points where they're sorted by y coordinates. So we're going to call those copies of points px, that's an array of the points sorted by x coordinate, and py for them sorted by y coordinate. Now, we know merge short takes n log n times, so this preprocessing step only takes o of n log n time. And again, given that we're shooting for an algorithm with running time big O of n log n, why not sort the points? We don't even know how we're going to use this fact right now, but it's sort of harmless. It's not going to effect our goal of getting a big of O n log n time algorithm. And indeed, this illustrates a broader point, which is one of the themes of this course. So recall, I hope one of the things you take away from this course is a sense for what are the four free primitives, what are manipulations or operations you can do on data which basically are costless. Meaning that if your data set fits in the main memory of your computer, you can basically invoke the primitive and it's just going to run blazingly fast, and you can just do it even if you don't know why. And again, sorting is the canonical for free primitive, although, we'll see some more later in the course and so, here, we're using exactly that principle. So we don't even understand why yet we might wa nt the points to be sorted. It just seems like it's probably going to be useful, motivated by the 1D case, so let's go ahead and make assorted copies of the points by x and y coordinate upfront. So reasoning by analogy with the 1D suggests that sorting the points might be useful, but we can't carry this analogy too far. So in particular, we're not going to be able to get away with just a simple linear scan through these arrays to identify the closest pair of points. So, to see that, consider the following example. So we're going to look at a point set which has six points. There's going to be two points, which I'll put in blue which are very close in x coordinate, but very far away in y coordinate. And then there's going to be another pair of points which I'll do in green, which are very close in y coordinate, but very far away in x coordinate. And then there's going to be a red pair of points, which are not too far away in either the x coordinate or the y coordinate. So in this set of six points, the closest pair is the pair of red points. They're not even going to show up consecutively on either of the two arrays, right? So in the array that's sorted by x coordinate, this blue point here is going to be wedged in between the two red points, they won't be consecutive. And similarly in the, in py, which is sort of by y coordinate, this green coordinate is going to be wedged between the two red points. So you won't even notice these red point if you just do a linear scan if your px and py, or py look at the consecutive pairs of points. So, following our preprocessing step where we just invert, invoke merge sort twice we're going to do a quite nontrivial divide and conquer algorithm to compute the closest pair. So really, in this algorithm, we're applying the divide and conquer algorithm twice. First, internal to the sorting subroutine, assuming that we use the merge sort algorithm to sort. Divide and conquer is being used there to get an n log n running time in this preprocessing step, and the n, we're going to use it again on sorted arrays in a new way and that's what I'm going to tell you about next. So let's just briefly review the divide and conquer algorithm design paradigm before we apply it to the closest pair problem. So, as usual, the first step is to figure out a way to divide your problem into smaller subproblems. Sometimes this has a reasonable amount of ingenuity, but it's not going to. Here in the closest pair problem, we're going to proceed exactly as we did in the merge sort and counting inversions problems, where we took the array and broke it into its left and right half. So here, we're going to take the input point set, and again, just recurse on the left half of the points, and recurse on the right half of the points. So here, by left and right, I mean with respect to the points x coordinates. There's pretty much never any ingenuity in the conquer step, that just means you take the sub-problems you identified in the first step, and you solve them recursively. That's what we'll do here, we'll recursively complete the closest pair in the left half of the points, and the closest pair in the right half of the points. So where all the creativity in divide and conquer algorithms is in the combined step. Given the solutions to your sub problems, how do you somehow recover a solution to the original problem? The one that you actually care about. So for closest pair, the questionis going to be, given that you've computed the closest pair on the left half of the points, and the closest pair on the right half of the points, how do you then quickly recover the closest pair for the whole point set? That's a tricky problem, that's what we're going to spend most of our time on. So let's make this divide and conquer approach for closest pair a little bit more precise, so let's now actually start spelling out our closest pair algorithm. The input we're given, it's, this follows the preprocessing steps or recall that we invoke, merge sort, we get our two sorted copies of the poin t set Px, sorted by x coordinate, and py sorted by y coordinate. So the first dividend is the division step. So given that we have a copy of the points px sorted by x coordinate, it's easy to identify the leftmost half of the points, those with the, those n over two smallest x coordinates and in the right half, those were the n over two largest x coordinates. We're going to call those Q and R respectively. One thing I'm skipping over is the base case. I'm not going to bother writing that down, so base case omitted, but it's what you would think it would be. So basically once you have a small number point, say two points or three points, then you can just solve the problem in constant time by a brute-force search. You just look at all the pairs and you return the closest pair. So think of it being at least four points in the input. Now, in order to recurse, to call clo pair again, in the left and right halves, we need sorted version of Q and R, both by x coordinate and by y coordinate, so we're just going to form those by doing suitable linear scans through px and py. And so one thing I encourage you to think through carefully or maybe even code up after the video is how would you form Qx, Qy, Rx and Ry given that you already have Px and Py. And if you think about it, because Px and Py are already sorted just producing these sorted sublists takes linear time. It's in some sense the opposite of the merge subroutine used in merge sort. Here, we're sort of splitting rather than merging. But again, this can be done in linear time, that's something you should think through carefully later. So that's the division step, now we just conquer, meaning we recursively call closest pair line on each of the two subproblems, so when we invoke closest pair on the left half of the points on Q we're going to get back what are indeed, the closest pair of points amongst those in Q. So we're going to call those P1 and Pq, So among all pairs of points that both lie in Q, P1 and Q1 minimize the distance between them. Similarly, we're going to call Q2Q2 the results of the second recursive call, that is, P2 and Q2 are amongst all pairs of points that both lie in R, the pair that has the minimum Euclidean distance. Now, conceptually, there's two cases. There's a lucky case and there's an unlucky case. In the original point set P, if we're lucky, the closest pair of points in all of P, actually, both of them lie in Q or both of them lie in R. In this lucky case, we'd already be done if the closest pair in the entire point set they happen to both lie in Q, then this first recursive call is going to recover them and we just have them in our hands P1Q1. Similarly, if both of the closest pair of points in all of P lies on the right side in R, then they get handed to us on a silver platter by the second recursive call that just operate on R. So in the unlucky case, the closest pair of point in P happens to be split. That is, one of the points lies in the left half, in Q, and the other point lies in the right half, in R. Notice, if the closest pair of points in all of P is split, is half in Q and half in R, neither recursive call is going to find it. Okay? The pair of points is not passed to either of the two recursive calls, so there's no way it's going to be returned to us. Okay? So we have not identified the closest pair after these two recursive calls, if the closest pair happens to be split. This is exactly analagous to what happened when we were counting inversions. The recursive call on the left half of the array counted the left inversions. The recursive call on the right half of the array counted the right inversions. But we still had to count the split inversions, so in this closest pair algorithm, we still need a special purpose subroutine that computes the closest pair for the case in which it is split, in which there is one point in Q and one point in R. So just like in counting inversions, I'm going to write down that subroutine and I'm going to leave it unimplemented for now, we'll figur e out how to implement it quickly in the rest of the lecture. Now, if we have a correct implementation of closest split pair, so that takes us input the original point set sort of the x and y coordinate, and returns the smallest pair that's split or one points in Q and one points in R, then we're done. So then, the split, then the closest pair has to either be on the lef or onm the right or it has to be split. Steps two through four compute the closest pair in each of those categories, so those are the only possible candidates for the closest pair and we just returned the best of them. So that's an argument for y, if we have a correct implementation of the closest split para subroutine, then that implies a correct implementation of closest pair. Now, what about the running time? So the running time of the closest para algorithm is going to be in part determined by the running time of closest split pair. So in the next quiz, I want you to think about what kind of running time we should be shooting for with a closest split pair subroutine. So the correct response of this quiz is the second one, and the reasoning is just by analogy with our previous algorithms for merge sort and for counting inversions. So, what is all of the work that we would do in this algorithm or we do have this preprocessing step we call merge sort twice, we know that's n log n, so we're not going to have a running time better than n log n cause we sort at the beginning. And then, we have a recursive algorithm with the following flavor, it makes two recursive calls. Each recursive call is on a problem of exactly half the size with half the points of the original one. And outside of the recursive calls, by assumption, by, in the problem, we do a linear amount of work in computing the closest split pair. So we, the exact same recursion tree which proves an n log n bound for merge sort, proves an n log n bound for how much work we do after the preprocessing step, so that gives us an overall running time bound of n log n. Remem ber, that's what we were shooting for. We were working n log n already to solve the one-dimensional version of closest pair and the goal of these lectures is to have an n log n algorithm for the 2D versions. So this would be great. So in other words, the goal should be to have a correct linear time implementation of the closest split pair subroutine. If we can do that, we're home-free, we get the desired n log algorithm. Now, I'm going to proceed in a little bit to show you how to implement closest split pair, but before I do that, I want to point out one subtle, the key idea, which is going to allow us to get this linear time correct implementation. So, let me just put that on the slide. So, the key idea is that we don't actually need a full-blown correct implementation of the closets split pair subroutine. So, I'm not actually going to show you a linear time subroutine that always correctly computes the closets split pair of a point set. The reason I'm going to do that is that's actually a strictly harder problem than what we need to have a correct recursive algorithm. We do not actually need a subroutine that, for every point sets, always correctly computes the closest split pair of points. Remember, there's a lucky case and there's an unlucky case. The lucky case is where the closest pair in the whole point set P happens to lie entirely in the left half of the points Q or in the right half of the points R In that lucky case, we, one of our recursive calls will identify this closest pair and hand it over to us on a silver platter. We could care less about the split pairs in that case. We get the right answer without even looking at the split pair, pairs. Now, there's this unlucky case where the split pairs happens to be the closest pair of points. That is when we need this linear time subroutine, and only. then, only in the unlucky case where the closest pair of points happens to be split. Now, that's in some sense, a fairly trivial observation, but, there's a lot of ingenuity here i n figuring out how to use that observation. The fact that we only need to solve a strictly easier problem and that will enable the linear time implementation that I'm going to show you next. So now, let's rewrite the high level recursive algorithm slightly to make use of this observation that the closest split pair subroutine only has to operate correctly in the regime of the unlucky case, when in fact, the closest split pair is closer than the result of either recursive call. So I've erased the previous steps 4 and 5, that, but we're going to rewrite them in a second. So, before we invoke close split pair, what we're going to do is we're going to see how well did our recursive calls do. That is, we're going to define a parameter little delta, which is going to be the closest pair that we found or the distance of the closest pair we found by either recursive call. So the minimum of the distance between P1 and Q1, the closest pair that lies entirely on the left, and P2Q2, the closest pair that lies entirely on the right. Now, we're going to pass this delta information as a parameter into our closest split pair subroutine. We're going to have to see why on earth that would be useful and I still owe you that information, but, for now, we're just going to pass delta as a parameter for use in the closest split pair. And then, as before we just do a comparison between the three candidate closest pairs and return the best of the, of the trio. And so, just so we're all clear on, on where things stand, so what remains is to describe the implementation of closest split pair, and before I describe it, let me just be crystal clear on what it is that we're going to demand of the subroutine. What do we need to have a correct in o of n log n time closest pair algorithm. Well, as you saw on the quiz, we want the running time to be o of n always, and for correctness, what do we need? Again, we don't need it to always compute the closest split pair, but we need it to compute the closest split pair in the events that there is a split pair of distance strictly less than delta, strictly better than the outcome of either recursive call. So now that we're clear on what we want, let's go ahead and go through the pseudocode for this closest split pair subroutine. And I'm going to tell you upfront, iIt's going to be fairly straightforward to figure out that the subroutine runs in linear time, o of n time. The correctness requirement of closest split pair will be highly non-obvious. In fact, after I show you this pseudo you're not going to believe me. You're going to look at the pseudocode and you'd be like, what are you talking about? But in the second video, on the closest pair lecture, we will in fact show that this is a correct sub-routine. So, how does it work? Well, let's look at a point set. So, the first thing we're going to do is a filtering step. We're going to prune a bunch of the points away and so to zoom in on a subset of the points. And the subset of the points we're going to look at is those that lie in a vertical strip, which is roughly centered in the middle of the point set. So, here's what I mean. By center dot, we're going to look at the middle x coordinate. So, let x bar be the biggest x coordinate in the left half, so that is in the sorted version of the points by x coordinate, we look at the n over two smallest ex-coordinate. So, in this example where we have six points, all this means is we draw, we imagine drawing a line between the third points, so that's going to be x bar, the x coordinate of the third point from the left. Now, since we're passed as input, a copy of the points sorted by x coordinate, we can figure out what x bar is in constant time. Just by accessing the relevant entry of the array, px. Now, the way we're going to use this parameter delta that we're passed, so remember what delta is. So before we invoke the closest split pair subroutine in the recursive algorithm, we make our two recursive calls, we find the closest pair on the left, the closest pair on the right, and delta is whatever the smaller of those two distances are. So delta is the parameter that controls whether or not we actually care about the closest split pair or not, we care if and only if there is a split pair at distance less than delta. So, how do we use delta? Well, that's going to determine the width of our strip, so the strip's going to have width 2 delta, and it's going to be centered around x. And the first thing we're going to do is we're going to ignore, forevermore, points which do not line in this vertical strip. So the rest of the algorithm will operate only on the subset of p, the subset of the points that lie on the strip, and we're going to keep track of them sorted by y coordinate. So the formal way to say that they line the strip, is that they have x coordinate in the interval with lower endpoint x bar minus delta and upper endpoint x bar plus delta. Now, how long does it take to construct this set Sy sorted by y coordinate? Well fortunately, we've been passed as input a sorted version of the points Py So to extract Sy from Py, all we need to do is a simple linear scan through p y checking for each point where its x coordinate is. So this can be done in linear time. Now, I haven't yet shown you why it's useful to have this sorted set as y, but if you take it on faith that it's useful to have the points in this vertical strip sorted by y coordinate. You now see why it was useful that we did this merge sort all the way at the beginning of the algorithm before we even underwent any recurssion. Remember, what is our running time goal for closest split pair? We want this to run in linear time, that means we cannot sort inside the closest split pair subroutine. That would take too long. We want this to be in linear time. Fortunately, since we sorted once and for all at the beginning of the closest pair algorithm, extracting sorted sublists from those sorted lists of points can be done, done in linear time, which is within our goals here. Now, it's the rest of t he subroutine where you're never going to believe me that it does anything useful. So, I claim that essentially with a linear scan through Sy, we're going to be able to identify the closest split pair of points in the interesting, unlucky case where there is such a split pair with distance less than delta. So here's what I mean by that linear scan through Sy. So as we do the scan, we're, we're going to keep track of the closest pair of points of a particular type that we've seen so far. So, let me introduce some variables to keep track of the best candidate we've seen so far. There's going to be a vary, variable best which will initialize to be delta. Remember, we're uninterested in split pairs unless they have distance strictly less than delta. So, and then we're going to keep track of the points themselves, so we'll initialize the best pair to be null. Now, here is the linear scan. So we go through the points of Sy in order y coordinate. Okay, well, not quite all the points of Sy. We stop at the eighth to last point and you'll see why in a second. And then, for each position I of the array Sy, we investigate the seven subsequent points of the same array Sy. So for j going from one to seven, we look at the Ith, and I plus jth entry of Sy. So if sy looks something like this array here, in any given point in this double for loop, we're generally looking at an index I, a point in this, in this of the array, and then some really quite nearby point in the array I plus j, because j here's going to be at most seven. Okay? So we're constantly looking at pairs in this array, but we're not looking at all pairs of all. We're only looking at pairs that are very close to each other, within seven positions of each other. And what do we do for each choice of i and j? Well, we just look at those points, we compute the distance, we see if it's better than all of the pairs of points of this form that we've looked at in the past and if it is better, then we remember it. So we just remember the best, ie c losest pair of points, of this particular type for choices of i and j of this form. So in more detail, if the distance between the current pair of points of p and q is better than the best we've seen so far, we reset the best pair of points to be equal to p and q, and we reset the best distance, the closest distance seemed so far to be the distance between p and q and that's it. Then, once this double for loop terminates, we just return it the best pair. So one possible execution of closest split pair is that it never finds a pair of points, p and q, at distance less than delta. In that case, this is going to return null and then in the outer call. In the closet pair, obviously, you interpret a null pair of points to have an infinite distance. So if you call closest split pair, and it doesn't return any points, then the interpretation is that there's no interesting split pair of points and you just return the better of the results of the two recursive calls p1Q1 or P2Q2. Now, as far as the running time of the subroutine, what happens here? Well, we do constant work just initializing the variables. Then notice that the number of points in Sy, well in the worst case, you have all of the points of P. So, it's going to be the most endpoints, and so, you do a linear number of iterations in the outer for loop. But here is the key point, in the inner for loop, right, normally double for loops give rise to quadratic running time, but in this inner for loop we only look at a constant number of other positions. We only look at seven other positions and for each of those seven positions, we only do a constant number of work. Right? We just, we want to compare distance and make a couple other comparisons, and reset some variables. So for each of the linear number of outer iterations, we do a constant amount of work, so that gives us a running time of o of n for this part of the algorithm. So as I promised, analyzing the running time of this closest split pair subroutine was not challenging. We just , in a straightforward way, looked at all the operations. Again, because in the key linear scan, we only do constant work per index, the overall running time is big O of n, just as we wanted. So this does mean that our overall recursive algorithm will have running time o of n log n. What is totally not obvious and perhaps even unbelievable, is that this subroutine satifies the correctness requirements that we wanted. Remember, what we needed, we needed that whenever we're in the unlucky case, whenever, in fact, the closest pair of points in the whole point set is split, this subroutine better find it. So, but it does, and that's being precise in the following correctness claim. So let me rephrase the claim in terms of an arbitrary split pair, which has distance less than delta, not necessarily the closest such pair. So suppose, there exists, a p on the left, a point on the left side and a point on the right side so that is a split pair and suppose the distance of this pair is less than Q. Now, there may or may not be such a pair of points, PQ.. Don't forget what this parameter delta means. What delta is, by definition, is the minimum of d of p1q1, for p1q1 is the closest pair of points that lie entirely in the left half of the point set Q and d of p2q2, or similarly, p2Q2 is the closest pair of points that entirely on the right inside of R. So, if there's a split pair with distance less than delta, this is exactly the unlucky case of the algorithm. This is exactly where neither recursive call successfully identifies the closest pair of points, instead that closest pair is a split pair. On the other hand, if we are in the lucky case, then there will not be any split pairs with distance less than delta, because the closest pair lies either all on the left or on the right, and it's not split. But remember, we're interested in the case where there is a split pair that has a distance less than delta where there is a split pair that is the closest pair. So the claim has two parts. The first part, part A, says the following. It says that if there's a split pair p and, and q of this type, then p and q are members of Sy. And let me just sort of redraw the cartoon. So remember what Sy is. Sy is that vertical strip. And again, the way we got that is we drew a line through a median x coordinate and then we fattened it by delta on either side, and then, we focused only on points that lie in the vertical strip. Now, notice our counts split pair subroutine, if it ever returns a pair of points, it's going to return a pair of points pq that belong to Sy. First, it filters down to Sy, then it does a linear search through Sy. So if we want to believe that our subroutine identifies best split pairs of points, then, in particular, such split pairs of points better show up in Sy, they better survive the filtering step. So that's precisely what part A of the claim is. Here's part B of the claim and this is the more remarkable part of the claim, which is that p and q are almost next to each other in this sorted array, Sy. So they're not necessarily adjacent, but they're very close, they're within seven positions away from each other. So, this is really the remarkable part of the algorithm. This is really what's surprising and what makes the whole algorithm work. So, just to make sure that we're all clear on everything, let's show that if we prove this claim, then we're done, then we have a correct fast implementation of a closest pair algorithm. I certainly owe you the proof of the claim, that's what the next video is going to be all about, but let's show that if the claim is true, then, we're home-free. So if this claim is true, then so is the following corollary, which I'll call corollaryl 1. So corollary 1 says, if we're in the unlucky case that we discussed earlier, if we're in the case where the closest point and the whole points of p does not lie both on the left, does not lie both on the right, but rather has one point on the left and one on the right but as it's a split pair, th en in fact, the count split pair subroutine will correctly identify the closest split pair and therefore the closest pair overall. Why is this true? Well what does count split pair do? Okay, so it has this double for loop, and thereby, explicitly examines a bunch of pairs of points and it remembers the closest pair of all of the pairs of points that it examines. What does this, so what are the criteria that are necessary for count split pair to examine a pair point? Well, first of all, the points p and q both have to survive the filtering step and make it into the array Sy. Right? So count split pair only searches over the array Sy. Secondly, it only searches over pairs of points that are almost adjacent in Sy, that are only seven positions apart, but amongst pairs of points that satisfy those two criteria, counts but pair will certainly compute the closest such pair, right? It just explicitly remembers the best of them. Now, what's the content of the claim? Well, the claim is guaranteeing that every potentially interesting split pair of points and every split pair of points with distance less than delta meets both of the criteria which are necessary to be examined by the count split pair subroutine. So first of all, and this is the content of part A, if you have an interesting split pair of points with distance less than delta, then they'll both survive the filtering step. They'll both make it into the array Sy., part A says that. Part B says they're almost adjacent in Sy. So if you have an interesting split pair of points, meaning it has distance less than delta, then they will, in fact, be at most seven positions apart. Therefore, count split pair will examine all such split pairs, all split pairs with distance less than delta, and just by construction, it will compute the closest pair of all of them. So again, in the unlucky case where the best pair of points is a split pair, then this claim guarantees that the count split pair will compute the closest pair of points. Therefore, having h andled correctness, we can just combine that with our earlier observations about running time and corollary 2 just says, if we can prove the claim, then we have everything we wanted. We have a correct O of n log n implementation for the closest pair of points. So with further work and a lot more ingenuity, we've replicated the guarantee that we got just by sorting for the one-dimensional case. Now again, these corrollaries hold only if this claim is, in fact, true and I have given you no justification for this claim. And even the statement of the claim, I think, is a little bit shocking. So if I were you I would demand an explanation for why this claim is true, and that's what I'm going to give you in the next video.

# 3-5-O(n log n) Algorithm for Closest Pair II [Advanced - Optional]

Alright. So the plan for this video is to prove the correctness of the divide and conquer closest to pair algorithm that we discussed in the previous video. So just to refresh your memory, how does the outer algorithm work? Well, we're given endpoints in the plane. We begin by sorting them, first by x-coordinate and then by y-coordinate. That takes n log in time. Then we enter the main recursive divide and conquer part of the algorithm. So what do we do? We divide the point set into the left half and the right half, Q and R, then we conquer. We recursively compute the closest pair in the left half of the point set Q. We recursively compute the closest pair in the right half of the point set R. There is a lucky case where the closest pair on the entire point set lies either all on the left or all on the right. In that case, the closest pair is handed to us on a silver platter, by one of the two recursive calls. But there remains the unlucky case where the closest pair is actually split with one point on the left and one point on the right. So to get our N log N running time bound, analogous to Merge Short in our inversion counting, we need to have a linear time implementation of a subroutine which computes the best, the closest pair of points, which is split, one on the left and one on the right. Well, actually, we don't need to do quite that. We need to do something only a little bit weaker. We need a linear time algorithm, which whenever the closest pair in the whole point set is in fact split, then computes that split pair in linear time. So let me now remind you of how that subroutine works. So it has two basic steps. So first, there's a filtering step. So it looks at, first of all, a vertical strip, roughly down the middle of the point set. And it looks at, only at points which fall into that vertical strip. That was a subset of the points that we called S sub Y, 'cause we keep track of them sorted by Y coordinate. And then we do essentially a linear scan through SY. So we go through the points one at a time, and then, for each point, we look at only the almost adjacent points. So for each index I, we look only at J's that are between one and seven positions further to the right, than I. So among all such points, we compare them, we look at their distances. We remember the best such pair of points. And then that's what we return from the count split pair subroutine. So we've already argued, in the previous video, that the overall running time of the algorithm is N log N. And what remains to prove correctness. And we also argued, in the previous video, that correctness boils down to the following correctness claim. In the sense that, if we can prove this claim, then the entire algorithm is correct. So this is what remains. Our residual work is to provide a proof of the correctness claim. What does it say? It says consider any split pair that is one point p from the left side Q, capital Q, and another point little q drawn from the right side of the point set capital R. And fur, further suppose that it's an interesting split pair meaning that the distance between them's at most delta. Here delta is recall the parameter pass to the count split pair subroutine, which is the smallest distance between a pair of points all on the left or all on the right. And this is the only case we're interested in. There's two claims. First of all, for p and q, both members of the split pair survive the filtering step. They make it into the sorted list S sub Y, and second of all, they will be considered by that double for loop, in the sense that the positions of p and q in this array, S sub Y, differ by at most seven. So that's the story so far. Let's move on to the proof. So let's start with part A which is the easy part relatively of the claim. So remember what we start with, our assumptions. We have a point p, let's write it out in terms of the X coordinates, X1 and Y1, which is from the left half of the point set. And we have a point q, which we'll call X2Y2, which comes from the right half of the point set. And furthermore, we're assuming that these points are close to each other. And we're gonna use that hypothesis over and over again. So the Euclidean distance between p and q is no more than this parameter delta. So, first, something very simple, which is that if you have two points that are close in Euclidean distance, then both of their coordinates have to be close to each other, right? If you have two points, and they differ by a lot in one coordinate, then the Euclidean distance is gonna be pretty big as well. So, specifically. By our hypothesis, that p and q have Euclidean distance less than delta, it must be that the difference between their coordinates in absolute value is no more than delta, and as well, the difference between their y-coordinates is at most delta. Okay, and this is easy to see if you'd just return to the definition of Euclidean distance that we reviewed at the beginning of the discussion of closest points. Okay? So if your distance is at most delta, then in each coordinate, you differ by at most delta as well. Now, what does A say? [sound]. So proof of A. So what does part A of the claim assert? It asserts that p and q are both members of SY, are both members of that vertical strip. So another way of saying that is that the X coordinates of p and q, that is, the numbers X1 and X2 both are within delta of Xbar. Remember, Xbar was in some sense the median X coordinate. So the X coordinate of the N over two'th leftmost point. So we're gonna do a proof by picture, so consider, forget about the Y coordinates, that's of irrelevant right now, and just focus on the X coordinates of all of these points. So on the one hand we have X bar. This is the X coordinate of the N over two'th point to the left. And then there are the X coordinates which define the left and the right borders of that vertical strip. Namely Xbar-delta and Xbar+delta. And then somewhere in here are X1 and Y1, the X coordinates of the points we care about, p and q. So a simple observation, so because p comes from the left half of the point set, and Xbar is the rightmost X coordinate of the left half of the point set, the X coordinate is at most Xbar. Right? So all points of Q have X coordinate, at most, Xbar, in particular, p does. Similarly, since Xbar is the rightmost edge of the left half of the point set, everything in the right half of the point set has X coordinate, at least Xbar. So in particular, little q does as well. So what does this mean. So this means x1, wherever it is, has to be at the left of x bar. X2 wherever it is has to be to the right of x bar. What we're trying to prove is that they're wedged in between x bar minus delta and x bar plus delta. And the reason why that's true is because their x coordinates also differ by at most delta. Okay, so what you should imagine is. You can imagine x1 and x2 are sort of people tied by a rope at the waist. And this rope has length delta. So wherever x1 and x2 move, they're at most delta apart. Furthermore x1, we just observed, can't move any farther to the right than Xbar. So even if X1 moves as far to the right as it can, all the way to Xbar, X2, since it's at most delta away, tied by the waist, can't extend beyond X bar+ delta. By the same reasoning, X2 can't move any further to the left than Xbar, X1 being tied to the waist to X2, can never drift further to the left than Xbar minus delta. So that's the proof that X1 and X2 both lie within this region, and that defines the vertical strip. So that's part A. If you have any split pair whose distance between them is less than delta, they both have to wind up, in this vertical strip. And therefore wind up in the filtered set, the proof set, S sub Y. So that's part A of the claim. Let's now move to Part B. Recall what part B asserts. It says that the points p and q, this split pair that are distance only delta apart. Not only do they wind up in this sort of filtered set SY, but in fact, they are almost adjacent in SY, in the sense that the indices in the array differ by, at most, seven positions. And this is a part of the claim that is a little bit shocking. Really what this says is that we're getting away with more or less a variant of our one dimensional algorithm. Remember when we wanted to find the closest pair of points on the line, all we had to do was sort them by their single coordinate and then look at consecutive pairs and return the best of those consecutive pairs. Here what we're saying is really, once we do a suitable filtering focus on points in this vertical strip, then we just go through the points according to their Y coordinate. And okay, we don't just look at adjacent pairs. We look at pairs within seven positions, but still we basically do a linear sweep through the points in SY. According to their Y coordinate and that's sufficient to identify the closest split pair. So why on earth will this be true. So our workhorse in this argument will be a picture which I am going to draw on next. So I'm going to draw eight boxes, which have a height and width delta over two. So here, delta is the same parameter that gets passed to the closest split pair subroutine. And it's also the same delta which we're assuming p and q are closer to each other than, right? So that's, remember, that's one of our hypotheses in this claim. The distance between p and q is strictly less than delta. So we're gonna draw eight delta over two boxes. And they're gonna be centered at x bar. So, this same center of the vertical strip that defines S Y. And the bottom is going to be the smaller of the Y-coordinates of the points p and q. So it might be p, it might be q. It doesn't really matter. But the bottom is going to be the smaller of the two. So the picture then looks as follows. So the center of these collections of eight boxes, X bar, the bottom is the minimum of Y1, Y2. We're gonna have two rows and four columns. And needless to say, we're drawing this picture just for the sake of this correctness proof, right? This picture is just a thought experiment in our head. We're just trying to understand why the algorithm works. The algorithm, of course, does not draw these boxes. The subroutine, the, closest split pair subroutine is just that pseudo code we saw in the previous video. This is just to reason about the behavior of that subroutine. Now looking ahead, I'll make this precise in two lemmas that are about to come up, what's going to be true is the following. So, either p or q is on this bottom line, right? So we define the bottom to be the lower Y coordinate of the two. So maybe, for example, q is the one that has the smaller Y coordinate, in which case, is gonna be, somewhere, say, down here. P, you remember, is from the left half of the point sets. So p is maybe gonna be here or something. And we're gonna argue that both p and q have to be in these boxes. Moreover, we're gonna argue that these boxes are sparsely populated. Every one contains either zero or one point of the array S sub Y. So, what we're gonna see is that there's at most eight points in this picture, two of which are p and q, and therefore, if you look at these points sorted by Y coordinate, it has to be that they're within seven of each other, the difference of indices is no more than seven. So, we're gonna make those two statements precise one at a time by the following two lemmas. Let's start with lemma one. Lemma one is the easy one. And it states that all of the points of S sub Y, which show up in between the Y coordinates of the points we care about p and q have to appear in this picture, they have to lie in one of these eight boxes. So we're going to argue this in two steps. First, we're going to argue that all such points have to have Y coordinates within the relevant range of this picture between the minimum of Y1 and Y2 and delta more than that, and secondly that they have to have X coordinates in the range of this picture, namely between X bar minus delta and X bar plus delta. So let's start with Y coordinates. So again, remember this key hypothesis we have, okay. We're dealing with a split pair p-q that are close to each other. The distance between X and Y is strictly less than delta. So the very first thing we did at the beginning of this proof is we said well, if their Euclidean distance is less than delta then they have to differ by at most delta in both of their coordinates, in particular in their Y coordinate. Now remember whichever is lower of p and q, whichever one has a smaller y coordinate is precisely at the bottom of this diagram. For example, if q is the one with the smaller y coordinate, it might be on the black line right here. So that means in particular x has y coordinate no more than the top part of this diagram. No more than delta bigger than q. And of course all points with Y coordinates in between them are equally well wedged into this picture. So that's why all points of SY with a Y coordinate between those of p and q have to be in the range of this picture, between the minimum of the two Y coordinates and delta more than that. Now what about horizontally? What about the X coordinates? Well this just follows from the definition of S sub Y. So remember, S sub Y are the points that fall into this vertical strip. How did we define the vertical strip? Well it had center Xbar, and then we fattened it by delta on both sides. So just by definition, if you're an SY, you've gotta have X coordinates in the range of this picture. X delta plus minus, sorry, xbar plus minus delta. So that completes the proof of the lemma. So this is not. This is just a lemma. So I'll use a lower case qed. Remember this is just a step toward proving the overall correctness claim. But this is a good step. And again, the way you think about this is it says we draw this boxes. We know that either p or q is at the bottom. The other one is going to be on the other side of the black line x bar but will be in some other box so perhaps maybe p is here and the lemma is saying that all the relevant points of SY have to be somewhere in this picture. Now remember in our double for loop we only search seven positions away, so the concern is that this is a sorta super highly populated collection of eight boxes. That's the concern, but that's not going to be the case and that's exactly what lemma two is going to say. Not only do the points between p and q in Y coordinates show up in this diagram, but there have to be very few. In particular, every box has to be sparse, with population either zero or one. So, let's move on to lemma two. So formally the claim is [sound], we have at most one point of the point set in each of these eight boxes. And this is finally where we use, in a real way, the definition of delta. This is where we finally get the payoff from our realization long ago, that when defining the closest split pair subroutine, we only really need to be correct in the unlucky case. In the case we're not handed the right answer by one of our recursive calls. We're finally gonna use that fact in a fundamental way. So we're gonna proceed by contradiction. So we're going to think about what happens if there are two points in a single box and from that we'll be able to derive a contradiction. So, call the points that wind up in the same box A and B. So, to the contrary, suppose A and B lie in the same box. So, maybe this is A here, and this is B here, at antipodal corners of this particular box. So from this supposition, we have two consequences. First of all. I claim that A and B lie on the same side of the point set. They're either both in the left side, Q or they're both in the right side, R. So why is this true? Well it's because every box lies either entirely on the left half of the point set or on the right half of the point set. Recall how we define x bar. X bar is the x coordinate of the right most point amongst the left half of the point set capital Q. So therefore points with x coordinate at most x bar have to lie inside the left half Q. Points with x coordinates at least x bar have to lie inside the right half of the point set capital R. So that would be like in this example. A and b both lie in a box which is to the right of x bar. So they both have to come in the right half of the point set capital R. This is one place we are using that there are no ties in X coordinate, so if there's a point with X, X coordinate or X bar, we can count it as part of the left half. So every box, by virtue of being either to the left of xbar or to the right of xbar, can only contain points from a common half of the point set. So that's the first consequence of assuming that you have two points in the same box. The second consequence is, because the boxes are small, the points gotta be close. So, if A and B co-habitate a box, how far could they be from each other? Well, the farthest they could be is like I've drawn in the picture, with the points A and B. Where they're at opposite corners of a common box. And then you bust out Pythagorean's Theorem, and what do you get? You get that the distance between them is delta over two, the side of the box times root two. And what's relevant for us is this is strictly less than delta. Okay? But, now, here is where we use, finally, the definition of delta. Consequences one and two in tandem, contradict how we define delta. Remember what delta is. It's as close as two pair of, a pair of points can get if they both lie on the left side of the point set, or if they both lie on the right side of the point set. That is how we defined it. As small as a pair of points on a common half can get to each other. But what have we just done? We've exhibited a pair A and B that lie on the same half of the point set, and are strictly closer than delta. So that contradicts the definition of delta. So that completes the proof of lemma two. Let me just make sure we're all clear on why having proved lemma one and lemma two we're done with the proof part B of the claim and therefore the entire claim because we already proved part one, now a long time ago. So let's interpret the 2 lemmas in the context of our picture that we had all throughout. In terms of the eight boxes of side length delta over two by delta over two. So again, whichever is the lower of p and q, and again let's just for the sake of concreteness say it's q, is at the bottom of the picture. The other point is on the other half of the line Xbar, and is in one of the other boxes. So, for example, maybe p is right here. So lemma one says that every relevant point, every point that survives the filtering and makes it into SY, by virtue of being in the vertical strip, has to be in one of those boxes, okay? If it has Y coordinate in between p and q. Lemma two says that you can only have one point in each of these boxes from the point set, so that's gonna be at most eight total. [sound] So combining them. Lemmas one and two imply, that there are almost eight points in this picture and that includes p and q because they also occupy two of eight boxes. So in the worst case, if this is as densely populated as could possibly be, given lemmas one and two, every other box might have a point and perhaps every one of those points has a Y coordinate between p and q. But this is as bad as it gets. Any point of the strip with Y coordinate between p and q occupies a box. So, at most, there are these six wedged in between them. What does this mean? This means if from q you look seven positions ahead in the array, you are guaranteed to find this point p. So a split pair with distance less than delta is guaranteed to be identified by our double for loop. Looking seven positions ahead in the sorted array SY is sufficient to identify, to look at every conceivably interesting split pair. So that completes the assertion B of the correctness claim and we're done. That establishes that this supremely clever divide and conquer algorithm is indeed a correct O(nlog(n)) algorithm that computes the closest pair of a set of n points in the plane.

# 4-1-Motivation

In this series of videos we'll study the master method. Which is a general mathematical tool for analyzing the running time of divide and conquer algorithms. We'll begin in this video motivating the method. Then we'll give its formal description. That'll be followed by a video working through six examples. Finally we'll conclude with three videos that discuss proof of the master method. With a particular emphasis on the conceptual interpretation of the master method's three cases. So let me say at the outset that this lecture's a little bit more mathematical than the previous two. But it's certainly not just math for math's sake. We'll be rewarded for our work with this powerful tool, the master method, which has a lot of predictive power. It'll give us good advice about which divide and conquer algorithms are likely to run quickly and which ones are likely to run less quickly. Indeed it's sort of a general truism that novel algorithmic ideas often require mathematical analysis to properly evaluate. This lecture will be one example of that truism. As a motivating example consider the computational problem of multiplying two n digit numbers. Recall from our first set of lectures that we all learned the iterative grade school multiplication algorithm. And that that requires a number of basic operations, additions and multiplications, between single digits. Which grows quadratically with the number of digits, n. On the other hand we also discussed an interesting recursive approach using the divide and conquer paradigm. So recall, divide and conquer necessitates identifying smaller sub-problems. So for integer multiplication, we need to identify smaller numbers that we want to multiply. So we proceed in the obvious way, breaking each of the two numbers into its left half of the digits, and its right half of the digits. For convenience, I'm assuming that the number of digits n is even, but it really doesn't matter. Having decomposed x and y in this way, we can now expand the product and see what we get. So let's put a box around this expression and call it \*. So we begin with the sort of obvious recursive algorithm where we just evaluate the expression \* in the straightforward way. That is, \* contains four products involving n over two digit numbers, ac, ad, bc, and bd. So we make four recursive calls to compute them. And then we complete the evaluation in the natural way. Namely, we append 0s as necessary, and add up these three terms to get the final result. The way we reason about the running time of recursive algorithms like this one is using what's called a recurrence. So to introduce a recurrence, let me first make some notation T(n). This is going to be the quantity that we really care about. The quantity that we want to upper bound. Namely this will be the worst case number of operations that this recursive algorithm requires to multiply two n-digit numbers. This is exactly what we want to upper bound. A recurrence, then, is simply a way to express T(n) in terms of T of smaller numbers. That is the running time of an algorithm in terms of the work done by its recursive calls. So every recurrence has two ingredients. First of all it has a base case describing the running time when there's no further recursion. And in this integer multiplication algorithm, like in most divide and conquer algorithms, the base case is easy. Once you get down to a small input, in this case two one digit numbers, then the running time is just constant. All you do is multiply the two digits and return the result. So I'm going to express that by just declaring the T(1), the time needed to multiply one digit numbers, as bounded above by a constant. I'm not going to bother to specify what this constant is. You can think of it as one or two if you like. It's not going to matter for what's to follow. The second ingredient in a recurrence is the important one. And it's what happens in the general case when you're not in the base case, and you make recursive calls. And all you do is write down the running time in terms of two pieces. First of all the work done by the recursive calls, and second of all the work that's done right here now. Work done outside of the recursive calls. So on the left hand side of this general case, we just write T(n). And then we want an upper bound on T(n) in terms of the work done by recursive calls and the work done outside of recursive calls. And I hope it's self evident what the recurrence should be in this recursive algorithm for integer multiplication. As we discussed, there are exactly four recursive calls. And each is invoked on a pair of n/2 digit numbers. So that gives us 4 times the time needed to multiply n/2 digit numbers. So what do we do outside of the recursive call? Well, we pad the results of the recursive calls with a bunch of zeros and we add them up. And I'll leave it to you to verify that grade school addition, in fact, runs in time linear in the number of digits. So putting it all together the amount of work we do outside of the recursive calls is linear. That is, it's O(n). Let's move on to the second, more clever, recursive algorithm for integer multiplication, which dates back to Gauss. Gauss's insight was to realize in the expression \* that we're trying to evaluate, there's really only three fundamental quantities that we care about. The coefficients for each of the three terms in the expression. So this, leads us to hope that perhaps we can compute these three quantities using only three recursive calls, rather than four. And indeed, we can. So what we do is we recursively compute a times c, like before, and b times d, like before. But then we compute the product of a + b with c + d. And the very cute fact is if we number these three products, one, two, and three. That the final quantity that we care about, the coefficient of the 10 to the n/2 term, namely ad + bc. Is nothing more than the third product minus each of the first two. So that's the new algorithm, what's the new recurrence? The base case obviously is exactly the same as before. So the question then is, how does the general case change? And I'll let you answer this in the following quiz. So the correct response for this quiz is the second one. Namely the only thing that changes with respect to the first recurrence is that the number of recursive calls drops from four down to three. A couple of quick comments. So first of all, I'm being a little bit sloppy when I say there's three recursive calls, each on numbers with n/2 digits. When you take the sums a + b and c + d, those might well have n/2 plus 1 digits. Amongst friends, let's ignore that. Let's just call it n/2 digits in each of the recursive calls. As usual, the extra plus one is not going to matter in the final analysis. Secondly I'm ignoring exactly what the constant factor is in the linear work done outside of the recursive calls. Indeed it's a little bit bigger in Gauss's algorithm than it is in the naive algorithm with four recursive calls. But it's only by a constant factor. And that's going to be suppressed in the big O notation. Let's look at this recurrence and compare it to two other reccurrences, one bigger, one smaller. So first of all, as we noted, it differs from the previous recurrence of the naive recursive algorithm in having one fewer recursive call. So, we have no idea what the running time is on either of these two recursive algorithms. But we should confident that this one certainly can only be better. That's for sure. Another point of contrast is merge sort. So think about what the recurrence would look like for the merge sort algorithm. It would be almost identical to this one except instead of a three we'd have a two. Right? Merge sort makes two recursive calls, each on an array of half the size. And outside of the recursive calls it does linear work, namely for the merge sub-routine. We know the running time of merge sort. It's n log n. So this algorithm, Gauss's algorithm, is going to be worse, but we don't know by how much. So while we have a couple clues about what the running time of this algorithm might be more or less than. Honestly we have no idea what the running time of Gauss's recursive algorithm for integer multiplication really is. It is not obvious. We currently have no intuition for it. We don't know what the solution to this recurrence is. But it will be one super-special case of the general master method, which we'll tackle next.

# 4-2-Formal Statement

So having motivated and hyped up the generality of the master method and its use for analyzing recursive algorithms, let's move on to its precise mathematical statement. Now, the master method is, in some sense, exactly what you want. It's what I'm going to call a black box for solving recurrences. Basically, it takes an input a recurrence in a particular format and it spits out as output a solution to that recurrence, an upper bound on the running time of your recursive algorithm. That is, you just plug in a few parameters of your recursive algorithm, and boom, out pops its running time. Now, the master method does require a few assumptions, and let me be explicit about one of them right now. Namely, the master method, at least the one I'm going to give you, is only going to be relevant for problems in which all of the subproblems have exactly the same size. So for example, in merge sort, there are two recursive calls, and each is on exactly one half of the array. So merge sort satisfies this assumption, both subproblems have equal size. Similarly, in both of our integer multiplication algorithms, all subproblems were on integers with n over 2 digits, with half as many digits, so those will all also obey this assumption. If for some reason you had a recursive algorithm that recursed on a third of the array and then on the other two-thirds of the array, the master method that I'm going to give you will not apply to it. There are generalizations of the master method that I'm going to show you which can accommodate unbalanced subproblem sizes, but those are outside the scope of this course. This will be sufficient for almost all of the examples we're going to see. One notable exception, for those of you that watched the optional video on a deterministic algorithm for linear time selection, that will be one algorithm which has two recursive calls on different subproblem sizes. So to analyze that recurrence, we'll have to use a different method, not the master method. Next I'm going to describe the format of the recurrences to which the master method applies. As I said, there are more general versions of the master method which apply to even more recurrences. But the one I'm going to give you is going to be reasonably simple, and it will cover pretty much all the cases you're likely to ever encounter. So recurrences have two ingredients. There's the relatively unimportant, but still necessary, base case step. And we're going to make the obvious assumption, which is just satisfied by every example we're ever going to see in this course, which is that at some point, once the input size drops to a sufficiently small amount, then the recursion stops, and the subproblem is solved in constant time. Since this assumption is pretty much always satisfied in every problem we're going to see, I'm not going to discuss it much further. Let's move on to the general case where there are recursive calls. So we assume the recurrence is given in the following format. The running time on an input of length n is bounded above by some number of recursive calls, let's call it a different recursive calls. And then each of these subproblems has exactly the same size, and it's 1 over b fraction of the original input size. So there's a recursive calls, each on an input of size n over b. Now, as usual, there's the case where n over b is a fraction and not an integer. And as usual, I'm going to be sloppy and ignore it. And as usual, that sloppiness has no implications for the final conclusion. Everything that we're going to discuss is true for the same reasons in the general case where n over b is not an integer. Now, outside the recursive calls, we do some extra work. And let's say that it's O(n to the d) for some parameter d. So in addition to the input size n, there are three letters here which we need to be very clear on what their meaning is. So first of all, there's a, which is the number of subproblems, the number of recursive calls. So a could be as small as 1 or it might be some larger integer. Then there's b. b is the factor by which the input size shrinks before a recursive call is applied. b is some constant strictly greater than 1. So for example, if you recurse on half of the original problem, then b would be equal to 2. It better be strictly bigger than 1 so that eventually you stop recursion, so that eventually then you terminate. Finally, there's d, which is simply the exponent in the running time of the, quote, unquote combine step, that is, the amount of work which is done outside of the recursive calls. And d could be as small as 0, which would indicate constant amount of work outside of the recursive calls. One point to emphasize is that a, b, and d are all constants. They're all numbers that are independent of n. So a, b, and d are going to be numbers like 1, 2, 3, or 4. They do not depend on the input size n. And in fact, let me just redraw the d so that you don't confuse it with the a. So again, a is the number of recursive calls and d is the exponent and the running time governing the work done outside of the recursive calls. Now, one comment about that final term, that big O(n to the d). On the one hand, I'm being sort of sloppy. I'm not keeping track of the constant that's hidden inside the big-O notation. I'll be explicit with that constant when we actually prove the master method. But it's really not going to matter. It's just going to carry through the analysis without affecting anything. So you can go ahead and ignore that constant inside the big-O. Obviously, the constant in the exponent, namely d, is very important. So depending on what d is, depends on whether that amount of time is constant, linear, quadratic, or so on. So certainly we care about the constant d. So that's the input to the master method. It is a recurrence of this form. So you can think of it as a recursive algorithm which makes a recursive calls, each on subproblems of equal size, each of size n over b, plus it does n to the d work outside of the recursive calls. So having set up the notation, I can now precisely state the master method for you. So given such a recurrence, we're going to get an upper bound on the running time. So the running time on inputs of size n is going to be upper bounded by one of three things. So somewhat famously, the master method has three cases. So let me tell you about each of them. The trigger, which determines which case you're in, is a comparison between two numbers. First of all, a, recall, a is the number of recursive calls made. And b raised to the d power. Recall, b is the factor by which the input size shrinks before you recurse. d is the exponent in the amount of work done outside of the recursive call. So we're going to have one case for when they're equal, we're going to have one case for when a is strictly smaller than b to the d. And the third case is when a is strictly bigger than b of the d. And in the first case, we got a running time of big O of n to the d times log n. And again, this is d, the same d that was in the final term of the recurrence. Okay, the work done outside of the recursive calls. So the first case, the running time is the same as the running time in the recurrence, outside of the recursive calls, but we pick up an extra log n factor. In the second case, where a is smaller than b to the d, the running time is merely big-O of n to the d. And this case might be somewhat stunning that this could ever occur, because of course, in recurrence, what do you do? You do some recursion, plus you do n to the d work outside of the recursion. So in the second case, it actually says that the work is dominated by just what's done outside the recursion in the outermost call. The third case will initially seem the most mysterious. When a is strictly bigger than b to the d, we're going to get a running time of big-O of n to the log base b of a. Where, again, recall, a is the number of recursive calls and b is the factor by which the input size shrinks before you recurse. So that's the master method with its three cases. Let me give this to you in a cleaner slide to make sure there's no ambiguity in my handwriting. So here's the exact same statement, the master method once again with its three cases, depending on how a compares to b to the d. So one thing you'll notice about this version of the master method is that it only gives upper bounds. So we only say that the solution to the recurrence is big-O of some function. And that's because if you go back to our recurrence, we used big-O rather than theta in the recurrence. And this is in the spirit of the course, where as algorithm designers, our natural focus is on upper bounds, on guarantees for the worst case running time of an algorithm. And we're not going to focus too much most of the time on proving stronger bounds in terms of theta notation. Now, a good exercise for you, to check if you really understand the proof of the master method after we go through it will be to show that if you strengthen the hypothesis and you assume the recurrence has the form T of n equals a times T of n over b plus theta of n to the d, then in fact, all three of these big-O's in the statement of the master method become thetas and the solution becomes asymptotically exact. So one final comment. You'll notice that I'm being asymmetrically sloppy with the two logarithms that appear in these formulas. So let me just explain why. In particular, you'll notice that in case one, with the logarithm, I'm not specifying the base. Why is that true? Well, it's because the the logarithm, with respect to any two different bases, differs by a constant factor. So the logarithm base e, that is, the natural logarithm, and the logarithm base 2, for example, differ by only a constant factor independent of the argument n. So you can switch this logarithm to whatever constant base you like, it only changes the leading constant factor, which of course is being suppressed in the big-O notation anyways. On the other hand, in case three, where we have a logarithm in the exponent, once it's in the exponent, we definitely care about that constant. Constants is the difference between, say, linear time and quadratic time. So we need to keep careful track of the logarithm base in the exponent in case three, and that base is precisely b, the factor by which the input shrinks with each recursive call. So that's the precise statement of the master method, and the rest of this lecture will work toward understanding the master method. So first, in the next video, we'll look at a number of examples, including resolving the running time of Gauss's recursive algorithm for integer multiplication. Following those several examples, we'll prove the master method. And I know now these three cases probably look super mysterious, but if I do my job, by the end of the analysis, these three cases will seem like the most natural thing in the world, as will these somewhat exotic looking formula for exactly what the running time is.

# 4-3-Examples

In this video, we'll put the master method to use by instantiating it for six different examples. But first, let's recall what the master method says. So the master method takes as input recurrences of a particular format, in particular recurrences that are parameterized by three different constants, A, B and D. A refers to the number of recursive calls, or the number of subproblems that get solved. B is the factor by which the subproblem size is smaller than the original problem size. And D is the exponent and the running time of the work done outside of the recursive calls. So the recurrence has the form, T(n), the running time on the input of size n, is no more than A, the number of subproblems, times the time required to solve each subproblem. Which is T(n/b) because the input size of a subproblem is n/b. Plus O(n to the d). The work outside of the recursive calls. There's also a base case which I haven't written down. So once the problem size drops below a particular constant then there should be no more recursion and you can just solve the problem immediately that is in constant time. Now given a recurrence in this permitted format, the running time is given by one of three formulas depending on the relationship between a, the number of recursive calls, and b raised to the d power. Case one of the master method is when these two quantities are the same, a = b to the d. Then the running time is n to the d log n, no more than that. In case 2, the number of recursive calls, a, is strictly smaller than b to the d. Then we get a better running time upperbound of O(n to the d). And when a is bigger than b to the d, we get this somewhat funky looking running time of O(n raised to the log base b of a power). We will understand where that formula comes from a little later. So that's the master method. It's a little hard to interpret the first time you see it. So let's look at some concrete examples. Let's begin with an algorithm that we already know the answer to, we already know the running time. Namely let's look at merge sort. So again what's so great about the master method is all we have to do is identify the values of the three relevant parameters A, B, and D, and we're done. We just plug them in and we get the answer. So a remembers the number of recursive calls. So in merge sort, recall we get two recursive calls. b is the factor by which the sub problem size is smaller than that in the original. Well we recurse on half the array. So the subproblem size is half that of the original. So b = 2. And recall that outside of the recursive calls, all merge sort does is merge. And that's a linear time subroutine. So the exponent D is 1, a reflection of the fact that it's linear time. So remember the key trigger which determines which of the three cases is the relationship between a and b to the d. So a obviously is 2. And b to the d = 2. So this puts us in Case 1. And remember in Case 1 we have that the running time is bounded above by O(n to the d (logn). In our case d = 1. So this is just O(n log n). Which, of course, we already knew. But at least this is a sanity check, the master method is at least reconfirming facts which we've already proven by direct means. So let's look at a second example. The second example is going to be for the binary search algorithm in a sorted array. Now we haven't talked explicitly about binary search and I'm not planning to, so if you don't know what binary search is, please read about it in a textbook or just look it up on the web, and it'll be easy to find descriptions. But the upshot it, this is basically how you'd look up a phone number in a phone book. Now I realize probably the youngest viewers of this video haven't actually had the experience of using a physical telephone book. But for the rest of you, as you know. You don't actually start with the As and then go to the Bs, and then go to the Cs if you're looking for a given name. You more sensibly split the telephone book roughly in the middle and depending if what you're looking for is early or later in the alphabet, you effectively recurse on the relevant half of the telephone book. So binary search is just exactly the same algorithm when you are looking for a given element in a particular sorted array. You start in the middle of the array, and then you recurse on the left or the right half as appropriate depending on if the element you're looking for is bigger or less than the middle element. Now the master method applies equally well to binary search and it tells us what its running time is. So in the next quiz you'll go through that exercise. So the correct answer is the first one. To see why let's recall what a, b, and d mean. a is the number of recursive calls. Now in binary search you only make one recursive call. This is unlike merge sort. Remember you just compare the element you're looking for to the middle element. If it's less than the middle element you recurse on the left half. If it's bigger than the middle element, you recurse on the right half. So in any case there's only one recursive call, so a is merely 1 in binary search. Now in any case you recurse on half the array so like in merge sort the value of b = 2, you recurse on a problem of half the size. And outside of the recursive column, the only thing you do is one comparison. You just determine whether the element you're looking for is bigger than or less than the middle element of the array that you recursed on. So that's constant time outside the recursive call giving us a value for d of 0. Just like merge sort, this is again Case 1 of the master method because we have a = b to the d, both in this case are equal to one. So this gives us a recurrence, a solution to our recurrence of big O(n to the d log n). Since d = 0 this is simply log n. And again many of you probably already know that the running time of binary search is log n or you can figure that out easily. Again this is just using the master method as a sanity check to reconfirm that it's giving us the answers that we expect. Let's now move on to some harder examples, beginning with the first recursive algorithm for integer multiplication. Remember this is where we recurse on four different products of n over two digit numbers and then re-combine them in the obvious way using adding by zero and some linear time additions. In the first integer multiplication algorithm, which, does not make use of Gauss's Trick where we do the four different recursive calls in a naive way, we have a, the number of recursive calls, = 4. Now in each case, whenever we take a product of two smaller numbers, the numbers have n over two digits so that's half as many digits as we started with. So just like in the previous two examples, b = 2. The input size drops by a factor 2 when we recurse. Now how much work do we do outside of the recursive call? Well again all it is doing is additions and adding by zeros and that can be done in linear time. Linear time corresponds to a primer value of d = 1. So next we determine which case of the master method we're in. a = 4, b to the d = 2, which in this case is less than a. So this corresponds to Case 3 of the master method. This is where we get the somewhat strange formula for the running time of the recurrence. T(n) = O(n to the log base b of a). Which with our parameter values, is n to the log base 2(4). Also known as O(n squared). So let's compare this to the simple algorithm that we all learned back in grade school. Recall that the iterative algorithm for multiplying two integers also takes an n squared number of operations. So this was a clever idea to attack the problem recursively. But at least in the absence of Gauss's Trick where you just naively compute each of the four necessary products separately. You do not get any improvement over the iterative algorithm that you learned in grade school. Either way, it's an n squared number of operations. But what if we do make use of Gauss's Trick, where we do only three recursive calls instead of four? Surely the running time won't be any worse than n squared, and hopefully it's going to be better. So I'll let you work out the details on this next quiz. So the correct answer to this quiz is the fourth option. It's not hard to see what the relevant values of a, b, and d are. Remember the whole point of Gauss's trick is to reduce the number of recursive calls from four down to three so the value of a is going to be 3. As usual we're recursing on a problem size which is half of that of the original. In this case n over two digit numbers so b remains 2. And just like in the more naive recursive algorithm, we only do linear work outside of the recursive call. So all that's needed to do some additions and patterns by 0. So that puts this parameter values a, b, and d. Then we have to figure out which case of the Master Method that is. So we have a = 3, b raised to the d = 2. So a has dropped by 1 relative to the more naive algorithm. But we're still in Case 3 of the Master Method. a is still bigger that the b to the d. So the running time is governed by that rather exotic looking formula. Namely T(n) = O(n to the log base b), which in our case is 2(a). Which is now 3 instead of 4, okay? So the master method just tells us the solution to this recurrence of 3. So what is log of the, what is log base 2(3)? Well plug it in your computer or your calculator, and you'll find that it's roughly 1.59. So we've got a running time of n to the 1.59. Which is certainly better than n squared. It's not as fast as n log n, not as fast as the merge sort recurrence, which makes only two workers for calls. But it's quite a bit better than quadratic. So summarizing, you did in fact learn a suboptimal algorithm for integer multiplication way back in grade school. You can beat the iterative algorithm using a combination of recursion plus Gauss's trick to save on the number of recursive calls. Let's quickly move on to our final two examples. Example number five is for those of you that watched the video on Strassen's matrix multiplication algorithm. So recall the salient properties of Strassen's algorithm. The key idea is similar to Gauss's Trick for integer multiplication. First you set up the problem recursively, one observes that the naive way to solve a problem recursively would lead to eight subproblems. But if you're clever about saving some computations, you can get it down to just seven recursive calls, seven subproblems. So a, in Strassen's argument, is equal to 7. As usual, each subproblem size is half that of the original one. So b = 2. And the amount of work done outside of the recursive calls is linear in the matrix size. So quadratic in the end, quadratic in the dimension because there's a quadratic number of entries in terms of the dimension. So n squared work outside the recursive call is leaving you a value of d = 2. So as far as which case of the master method we're in. Well it's the same as in the last couple of examples. a = 7, b to the d = 4 which is less than a. So once again we're in Case 3 and now the running time of Strassen's algorithm T(n) = O(n to the log base) 2(7) which is more or less n to the 2.81. And again this is a win. Once we use the savings to get down to just 7 recursive calls. This beats the naive federate algorithm, which recall would require cubic time. So that's another win for clever divide and conquer in matrix multiplication via Strassen's algorithm. And once again, the master's method just by plugging in parameters, tells us exactly what the right answer to this recurrence is. So for the final example I feel a little guilty because I've shown you five examples and none of the them have triggered Case 2. We've had two in Case 1 in the master method and three now in Case 3. So this will be a fictitious recurrence just to illustrate Case 2. But there are examples of recurrences that come up where Case 2 is the relevant one. So let's just look at the following recurrence. So this recurrence is just like merge sort. We recurse twice. There's two recursive calls each on half the problem size. The only difference is in this recurrence we're working a little bit harder on the combined step. Instead of linear time outside of the recursive calls we're doing a quadratic amount of work. Okay so a = 2. b = 2 and d = 2. So b to the d = 4, strictly bigger than a. And that's exactly the trigger for Case 2. Now recall what the running time is in Case 2. It's simply n to the d, where d is the exponent in the combined step. In our case, d is 2, so we get a running time of n squared. And you might find this a little counter-intuitive, right? Given that merge sort. All we do with merge sort is change the combine step from linear to quadratic. And merge sort has a running time of n log n. You might have expected the running time here to be n squared log n. But that would be an over estimate, so the master method gives us a tighter upper bound, shows that it's only quadratic work. So put differently, the running time of the entire algorithm is governed by the work outside of the recursive calls. Just in the outer most call to the algorithm, just at the root of the recursion tree

# 4-4-Proof I

In this video, we'll begin the proof of the master method. The master method, you'll recall, is a generic solution to recurrences of the given form, recurrences in which there's a recursive calls, each on a sub-problem of the same size, size n over b, assuming that the original problem had size n. And, plus, there is big O of n to the d work done by the algorithm outside of these a recursive calls. The solution that the master method provides has three cases, depending on how a compares to b to the d. Now. This proof will be the longest one we've seen so far by a significant margin. It'll span this video as well as the next two. So let me say a few words up front about what you might want to focus on. Overall I think the proof is quite conceptual. There's a couple of spots where we're going to have to do some computations. And the computations I think are worth seeing once in your life. I don't know that they're worth really committing to long term memory. What I do think is worth remembering in the long term however, is the conceptual meaning of the three cases of the master method. In particular the proof will follow a recursionary approach just like we used in the running time analysis of the Mertshot algorithm. And it worth remembering what three types of recursion trees the three cases Is that the master method corresponds to. If you can remember that, there will be absolutely no need to memorize any of these three running times, including the third, rather exotic looking one. Rather, you'll be able to reverse engineer those running times just from your conceptual understanding of what the three cases mean and how they correspond to recursion trees of different type. So, one final comment before we embark on the proof. So, as usual, I'm uninterested in formality in its own sake. The reason we use mathematical analysis in this course, is because it provides an explanation of, fundamentally, why things are the way they are. For example, why the master method has three cases, and what those three cases mean. So, I'll be giving you an essentially complete proof of the master method, in the sense that it has all of the key ingredients. I will cut corners on occasion, where I don't think it hinders understanding, where it's easy to fill in the details. So, it won't be 100 percent rigorous, I won't dot every I and cross every t, but. There will be a complete proof, on the conceptual level. That being said, let me begin with a couple of minor assumptions I"m going to make, to make our lives a little easier. So first, we're gonna assume that the recurrence has the following form. So, here, essentially, all I've done is I've taken our previous assumption about the format of a recurrence, and I've written out all of the constants. So, I'm assuming that the base case kicks in when the input size is one, and I'm assuming that the number of operations in the base case is at most c, and that, that constant c is the same one that was hidden in the big O notation of the general case of the recurrence. The constant c here isn't gonna matter in the analysis, it's just all gonna be a wash, but to make, keep everything clear, I'm gonna write out all the constants that were previously hidden in the big O notation. Another assumption I'm going to make. Now goes to our murtured analysis, is that N is a power of B. The general case would be basically the same, just a little more tedious. At the highest level, the proof of the master method should strike you as very natural. Really, all we're going to do is revisit the way that we analyze Merge Short. Recall our recursion tree method worked great, and gave us this [inaudible] log [inaudible], and the running time of Merge Short. So we're just gonna mimic that recursion tree, and see how far we get. So let me remind you what a recursion tree is. At the roots, at level zero, we have the outermost, the initial indication of the recursive algorithm. At level one, we have the first batch of recursive calls. At level two, we have the recursive calls made by that first batch of recursive calls, and so on. All the way down to the leaves of the tree, which correspond to the base cases, where there's no further recursion. Now, you might recall, from the Merge Sort analysis that we identified a pattern that was crucial in analyzing the running time. And that pattern that we had to understand was, at a given [inaudible] J, at a given level J of this recursion tree. First of all, how many distinct subproblems are there at level J? How many different level J [inaudible] are there? And secondly, what is the input size that each of those level J subproblems has to operate on? So think about that a little bit and give your answer in the following quiz. So the correct answer is the second one. At level J at. Of this recursion tree, there are A to, to the J sub-problems and each has an input of size of N over B to the J. So first of all, why are there A to the J sub-problems? Well, when J equals zero at the root, there's just the one problem, the original indication of the recursive algorithm. And then each. Call to the algorithm makes a further calls. For that reason the number of sub problems goes up by a factor of A with each level leading to A to the J sub problems at level J. Similarly, B is exactly the factor by which the input size shrinks once you makea recursive call. So J levels into the recursion. The input size has been shrunk J times by a fctor of B each time. So the input size at level J is N over B to the J. That's also the reason why, if you look at the question statement, we've identified the numbers of levels as being log of base B. Of N. Back in Merge Short, B was two. We [inaudible] on half the array. So the leaves all resided at level log base two of N. In general, if we're dividing by a factor B each time, then it takes a log based B of N times before we get down the base cases of size of one. So the number of levels overall, zero through log base B event. For a total of log based B event plus one levels. Here then is what the recursion tree looks like. At level zero we have the root corresponding to the outer most call. And the input size here is N. The original problem. Children of a node correspond to the recursive calls. Because there are A. Recursive calls by assumption, there are A. Children or A. Branches. Level one is the first batch of precursor calls. Each of which operates on an input of size N over B. That level log base B. Of N. We've cut the input size by a factor of B. This many times, so we're down to one. So that triggers the base case. So now, the plan is to simply mimic our previous analysis of Merge Sort. So let's recall how that worked. What we did is we zoomed in, in a given level. And for a given level J, we counted the total amount of work that was done at level J subproblems, not counting work that was gonna be done later by recursive calls. Then, given a bound on the amount of work at a given level J, we just summed up overall, the levels, to capture all of the work done by all of the, recursive indications of the algorithm. So inspired by our previous success let's zoom in on a given level J., and see how much work gets done, with level J. Sub problems. We're going to compute this in exactly the way we did in merge sort. And we were just going to look at the number of problems that are at level J and we're going to multiply that by a bound on the work done per sub-problem. We just identified the number of level J sub-problems as A to the J. To understand the amount of work done for each level j sub-problem, let's do it in two parts. So, first of all, let's focus on the size of the input for each level j sub-problem. That's what we just identified in the previous quiz question. Since the input size is being decreased by a factor b each time, the size of each level j sub-problem is n over b to the j. [inaudible] Now we only care about the size of a level J sub problem in as much it determines the amount of work the number of operations that we perform per level J sub problem. And to understand the relationship between those two quantities we just return to the re currents. The recurrent says how much work gets done in the specific sub problem well there's a bunch of work done by recursive calls the A recursive calls and we're not counting that we're just counting the work done here at A level J and the recurrence also tells us how much is done outside of the recurrent calls. Namely it's no more than the constant C times the input size. Raised to the d power. So here the input size is N over B to the J, so that gets multiplied by the constant C. And it gets raised to the D power. Okay. So C. Times quanity N. Over B. To the J. That's the emphasized. Raised to the D. Power. Next, I wanna simplify this expression a little bit. And I wanna separate out the terms which depend on the level number J, and the terms which are independent of the level number J. So if you look at it A and B are both functions of J, where the C and end of the D terms are independent of J. So let's just separate those out. And you will notice that we have now our grand entrance of the ratio between A and B to the D. And foreshadowing a little, recall that the three cases of the master method are governed by the relationship between A and B to the D. And this is the first time in the analysis where we get a clue that the relative magnitude of those two quantities might be important. So now that we've zoomed in on a particular label J and done the necessary computation to figure out how much work is done just at that level, let's sum over all of the levels so that we capture all of the work done by the algorithms. So this is just gonna be the sum of the epression we saw on the previous slide. Now since C into the D doesn't depend on J, I can yank that out in front of the sum, and I'll sum the expression over all J. That results in the following. So believe it or not, we have now reached an important milestone in the proof of the master method. Specifically, the somewhat messy looking formula here, which I'll put a green box around, is going to be crucial. And the rest of the proof will be devoted to interpreting and understanding this expression, and understanding how it leads to the three different running time bounds in the three different cases. Now I realize that at the moment this expression's star probably just looks like alphabet soup, probably just looks like a bunch of mathematical gibberish. But actually interpreted correctly this has a very natural interpretation. So we'll discuss that in the next video.

# 4-5-Interpretation of the 3 Cases

This video is the second of three that describes the proof of the Master Method. In the first of these three videos we mimicked the analysis of merge sort. We used a recursion tree approach which gave us an upper bound of running time of an algorithm. Which is governed by a recurrence of the specified form. Unfortunately, that video left us with a bit of an alphabet soup, this complicated expression. And so in this second video, we're not gonna do any computations. We're just going to look at that expression, attach some semantics to it, and look at how that interpretation naturally leads to three cases, and also give intuition for some of the running times that we see in a master method. So recall from the previous video that the way we've bounded the work done by the algorithm is resumed in on a particular level J of the recursion tree. We did a computation, which was the number of sub problems at that level, a to the j, times the work done per sub-problem, that was the constant C times quantity N over B to the J raised to the D and that gave us this expression. Cn to the D times the ratio of A over B to the D raised to the J. At a given level. J. The expression star that we concluded the previous video with was just the sum of these expressions over all of the logarithmic levels, J. Now, as messy as this expression might seem, perhaps we're on the right track in the following sense. The master method has three different cases, in which case you're in is governed by how A compares to B to the D. And hearing this expression, we are seeing precisely that ratio. A divided by B to the D. So let's drill down and understand why this ratio is fundamental to the performance of the divide and conquer [inaudible] algorithm. So really, what's going on in the master method, is a tug of war between two opposing forces. One which is forces of good, and one which is forces of evil, and those correspond to the quantities B to the D and A, respectively. So let me be more precise. Let's start with the parameter A. So A, you'll recall, is defined as the number of recursive calls made by the algorithm. So it's the number of children that a [inaudible] recursion tree has. So fundamentally, what A is, it's the rates at which sub problems proliferate as you pass deeper in the recursion tree. It's the factor by which there are more sub problems at the next level than the previous one. So let's think of A. In this way. As the rate of subpropabifliation, or R.S.P. And when I say rate I mean as a function of the recursion level J. So these are the forces of evil. This is why our algorithm might slowly, is because as we go down the tree there are more and more sub problems, and that's a little scary. The forces of good, what we have going for us, is that with each recursion level J we do less work per sub problem and the extent to which we do less work is precisely B to the D. So I'll abbreviate this rate of work shrinkage or this quantity B. To the D. By R. W. S. Now perhaps you're wondering why is it B of the D. Why is it not B? So remember what B denotes. That's the factor by which the input size shrinks with the recursion level J. So for example if B equals two, then each sub-problem at the next level is only half as big. As that at the previous level. But we don't really care about the input size of a subproblem, except inasmuch as it determines the amount of work that we do solving that subproblem. So that's where this parameter D comes into play. Think, for example, about the cases where you have a linear amount of work outside the recursive calls, versus a quadratic amount of work that is considered the cases where D equals one or two. If B = two and D = one that is if you reverse on half the input. And do linear work, then. Not only is the input size dropping by factor two but so is the amount of work that you do per sub problem and that's exactly the situation we had in merge short where we had linear work outside the recursive calls. The thing about D = two, suppose you did quadratic work per sub problem as a function of the input size. Then again if B = two if you cut the input in half, the recursive call's only gonna do 25 percent as much work as what you did. At the current level. The input size goes down by a factor two and that gets squared because you do quadratic work as a function of the input size. So that would be B to the D, two raised to the two or four. So in general the input size goes down by a factor B, but what we really care about, how much less work we do per subproblem, goes down by B to the D. That's why B to the D is the fundamental quantity that quan, that's governs the forces of good, the extent to which we work less hard with each occursion level J. So the question that is just what happens in this tug of war between these two opposing forces? So fundamentally, what the three cases of the master method correspond to, is the three possible outcomes in this tug of war between the forces of good, namely the rate of word shrinkage and the forces of evil, namely the sub-problem proliferation. There are three cases one for the case of a tie one for the case in which the forces of evil win that is in which A is bigger than B to the D and a case in which the forces of good wins, that is B to the D is bigger than A. To understand this a little bit better what I want you to think about is the following. Think about the recursion tree that we drew in the previous slide and as a function of A verses B to the D think about the amount of work you've done per level. When is that going up per level? When is it going down per level? And when is it exactly the same at each level? So the answer is all of these statements are true except for the third one. So let's take them one at a time. So first of all let's consider the first one. Suppose that the rate of sub problem proliferation A is strictly less than the rate of work Shrinkage, B to the D. This is where the forces of good, the rate at which we're doing less work per sub problem is out, out pacing the rate of at which sub problems are proliferating. And so the number of sub-problems goes up, but the savings per sub-problem goes up by even more. So, in this case it means that we're gonna be doing less work. With each recursion tree level, the forces of good outweigh the forces of evil. The second one is true for exactly the same reason. If sub-problems are proliferating so rapidly that it outpaces the savings that we get per sub-problem, then we're gonna see an increasing amount of work. As we go down the recursion tree, it will increase with the level of J. Given that these two are true the third one is false. We can draw conclusions depending on whether the rate of sub-problem proliferation is strictly bigger or strictly less than the rate of work shrinkage. And finally, the fourth statement is also true. This is the perfect equilibrium between the forces of good and the forces of evil. Sub-problems are proliferating, but our savings per sub-problem is increasing at exactly the same rate. The two forces will then cancel out and we'll get exactly the same amount of work done at each level of the recursion tree. This is precisely what happened when we analyzed a merd short algorithm. So let's summarize and conclude with the interpretation. And even understand how this interpretation lends us to forecast some of the running time bounds that we see in the Master Method. Summarizing, the three cases of the master method correspond to the three possible outcomes in the battle between sub-problems proliferating and the work per sub-problem shrinking. One for a tie, one for when sub-problems are proliferating faster, and one for when the work shrinkage is happening faster. In the case where the rates are exactly the same, and they cancel out, then the amount of work should be the same at every level of the recursion tree. And, in this case, we can easily predict what the running time should work out to be. In particular, we know there's a logarithmic number of levels, the amount of work is the same at every level, and we certainly know how much work is getting done at the root, right, because that's just the original recurrence, which tells us that there's, acentotically, n to the d work done at the root. So, with n to the d work for each of the log levels, we expect a running time of n to the d times log n. As we just discussed, when the rate of. Work done per subproblem is shrinking even faster than subproblems proliferate. Then we do less and less work with each level of the recursion tree. So in particular, the biggest amount of work, the worst level is at the root level. Now, the simplest possible thing that might be true would be that actually, the root level just dominates the overall running time of the algorithm, and the other levels really don't matter up to a constant factor. So it's not obvious that's true, but if we keep our fingers crossed and hope for the simplest possible outcome. With the root has the most work, we might expect a running time that's just proportional to the running time of the root. As we just discussed, we already know that that's n to the d, cuz that's just the outermost call to the algorithm. By the same reasoning, when this inequality is flipped, and [inaudible] proliferates so rapidly that it's outpacing the same as we get for sub problem, the amount of work is increasing the recursion level. And here, the worst case is gonna be at the leaves. That's where the, that level's gonna have the most work compared to any other level. And again, if you keep your fingers crossed and hope that the simplest possible outcome is actually true, perhaps the leaves just dominate, and. Up to a constant factor, they govern the running time of the algorithm. In this third case, given that we do a constant amount of work for each of the leaves, since those correspond to base cases, here we'd expect a running time in the simplest scenario, proportional to the number of leaves in the recursion tree. So lets summarize what we've learned in this video. We now understand that fundamentally there are three different kinds of recursion trees. Those in which the work done per level is the same in every level. Those in which the work is decreasing with the level in which case the root is the lowest level. And those in which the amount of work is increasing with the level where the leads are the lowest level. Further more it's exactly the ratio between A the rate of sub problem proliferation and B to the D the rate of work shrinkage sub problem That governs which of these three recursion trees we're dealing with. Further more. Intuitively, we've now had predictions about what kind of running time we expect to see in each of the three cases. They're N to the D log in, that we're pretty confident about. There's a hope that, in the second case, where the root is the worst level, that maybe the running time is N to the D. And there's a hope in the third case where the [inaudible] are the worse level, and we do constant time per leaf, per base case, that it's gonna be proportional to the number of leaves. Let's now stand and check this intuition against the formal statement of the master method, which we'll prove more formally in the next video. So in the three cases, we see they match up. At least two out of three with exactly [inaudible] lies. So in the first case, we see the expected end of the D times log in. In the second case, where the root is the worst level indeed, the simplest possible outcome of big O of N to the D is the assertion. Now, the third case that remains a mystery to be explained. Our intuition said this should hopefully be proportional to the number of leaves. And instead, we've got this funny formula of big O of N in the log base B of A. So in the next video, we'll demystify that connection, as well as supply formal proof for these assertions.

# 4-6-Proof II

Let's complete the proof of the master method. Let me remind you about the story so far, the first thing we did is we analyzed the work done by a recursive algorithm using a recursion tree. So we zoomed in on a given level J, we identified the total amount of work done at level J and then we summed up over all of the levels resulting in this rather intimidating expression star. C into the D times a sum over the levels J from zero to log base B of N of quantity A over B to the B raised to the J. Having derived this expression star we then spent some time interpreting it, attaching to it some semantics Sticks. And we realize that the roll of this ratio A to the B over D, is to distinguish between three fundamentally different types of recursion trees. Those in which A = B to the D and the amount of work is the same at every level. Those in which A is less than B to the D and therefore the amount of work is going down with the level. And those where A is bigger than B to the D in which case the amount of work is growing with the level. This gave us intuition about the three cases of the master method and even gave us predictions f or the running times we might see. So what remains to do is turn this hopeful intuition into. A rigorous proof. So we need to verify that in fact the simplest possible scenarios outlined in the previous video. Actually occur. In addition, we need to demystify the third case and understand what the expression has to do with the number of leaves of the recursion tree. Let's begin with the simplest case, which is case one. We're calling case one, we're assuming that A equals B to the D. This is the case where we have a perfect equilibrium between the forces of good and evil. Where the rate of the sub problem proliferation exactly cancels out with a rate at which we do less work per sub problem. And now, examining the expression, star, we can see how easy our lives get when A equals B to the D. In that case, this ratio is equal to one. So naturally this ratio raised to the J is also equal to one for all J. And then of course this sum evaluates to something very simple. Namely one summed with itself log base B of N plus one times. So the sum simply equals log base B of N. Plus one, and that's going to get multiplied by. This CN to the D term which is independent of the sum. So summarizing, when A equals B to the D, we find that star equals CN to the D times log base B of N plus one. Writing this in big o notation, we would write big o of end of a d login. And again, I'm going to suppress the base of the logarithms. Since all logarithms differ only by a constant factor we don't have to specify the base. That's just suppressed by the constant hidden in the big O notation. So that's it for case one. Like I said, this is the easy case. So what do we do when A is not equal to B to the D? And remember A could be either less than or bigger than B to the D. To answer that question, let's take a short detour into geometric series. For this single slide detour we're going to think about a single constant number R. Now, what you want to think about is R. Representing that ratio A. Over B. To the D. From the previous slot. But for this slide only let's just call it R. This is a constant. It's bigger than zero, and it's not equal to one. Now, suppose we sum up powers of R stopping, let's say, at the Kth power of R. I claim that this sum has a nice closed form formula. Specifically it is exactly, R. To the K. Plus one, minus one. Divided by or a minus one. Now, whenever you see a general formula like this, it's useful to keep in mind a couple of canonical values of the parameters that you can plug in to develop intuition. And for this expression, you might wanna think canonically about the cases, R=2, and R=1/2. So when R=2, or something that powers a two. One+2+4+8+16, and so on. One hour's a half, [inaudible] have one, plus a half, plus a quarter, plus an eighth, and so on. Now I'm not gonna prove this for you, I'd like you to prove this yourself. If you don't already know this fact. So the way to prove this is simply by induction. And I will leave this an, an exercise. What I wanna focus on instead is what this fact can do for us. The way that we use this fact is to formalize the idea that, that in recursion trees where the amount of work is increasing in the levels, the leaves dominate the overall running time. And where recursion trees, where the amount of work is decreasing in the level, the root dominates the running time. In the sense that we can ignore all of the other levels of the recursion tree. So, and in the vision in this slide, we have two upshots. First of all, for the case when R is less than one. And in this case, this expression on the right-hand side. R to the Q plus one minus one over R minus one can be upper bounded by one over one minus R. So again, remember, you might want to have a canonical value of r in mind here, namely, one half. So what we're claiming here is that the right hand side is nor more than two for the case of R=1/2. And that's easy to see if you think about one plus one-half plus a one-fourth plus one-eighth and so on. That sum is converging to, to as k grows large. So in general, for our less than one constant, the sum is divided by one minus one over R. Now, we're not actually gonna care about this formula, one minus one over R. The point for us is just that this is a constant. And by constant, I mean independent of K, independent of how many terms we sum up. Obviously, it depends on R of the ratio, but it does not depend on how many things we sum up on K. So the way to think about this is, when we sum up a bunch of terms where R is less than one, then the very first term dominates. The first term is with a one. And no matter how many terms we sum up, we never get, grow bigger than the sum constant. A similar situation holds for the case where r is a constant bigger than one. When r is bigger than one. A tiny bit of algebra shows that we can upper bound the right hand side by r to the k. Times something which is constant, independent of K. So again, let's interpret the second upshot in terms of a canonical value of R. Namely, R equals two. Then our sum is one plus two plus four plus eight plus sixteen, and so on. And what this is saying is that no matter how many terms we sum up, the overall sum is never gonna be more than twice. The largest and final term. So if we sum up to say 128, the sum, you'll notice, will be 255, which is, at most, twice that largest term, 128. And that saying is true for any K. The entire sum is no more than twice that of the largest term. In this sense, the largest term in the series dominates the whole thing. So to summarize this slide in just into one sentence we sum up powers of a constant R when R is bigger than one the largest power of that constant dominate to the sun when R is smaller than one then the sun is just a constant. Let's now apply this to prove case two of the master method. In case two of the master method, we assume that A is less than B to the D. That is, the rate at which sub problems are proliferating is drowned out by the rate at which we do less work per sub problem. So this is the case where the amount of work is decreasing with each level of the recursion tree. And our intuition said that, well, in the simplest possible scenario, we might hope that all of the work, up to a constant factor, is being done at the root. So let's make that intuition precise by using the basic Sums fact on the previous slide. So, since A is less than B to the D. This ration is less than one. So let's call this ratio equal to R. So R, you'll notice, does depend on the three parameters, A, B and D. But R is a constant, it does not depend on N. So what is this sum? The sum is just, we're just summing up powers of this constant R, where R is less than one. What did we just learn? We just learned that any such sum is bounded above by a constant, independent of the number of terms that you sum up. So therefore, what is this expression star evaluates to. It evaluates to C, which is a constant, times N to the D. Times another constant. So suppressing the product of these two constants in Big O notation we can say that the expression starts upper bounded by Big O(n to the d). And this makes precise our intuition that indeed the overall running time of the algorithm, in this type of recursion tree with decreasing work per level, is dominated by the root. The overall amount of work is only a constant factor larger than the work done and merely at level zero of the tree. Let's move on to the final and most challenging part of the proof, the final case. In case three we assume that A is bigger than B to the D. So in conceptual terms, we're assuming the rate at which sub problems proliferate is exceeding the rate at which we do less work per sub problem. So these are recursion trees where the amount of work is increasing with each level, with the most work being done at the leaves. And once again, using the basic sums fact, we can make precise the hope that, in fact, we only have to worry about the leaves. We can throw away the rest of work, losing only a constant factor. So to see that, you will again denote this ratio between A and B to the D as R. And in this case R is bigger that one. So this sum is a sum of a bunch of powers of R were R is bigger than one, what did we just learn about that two slides ago in the basic sums facts, we learned that such sums are dominated by the largest and last term of the sum. Okay so the bounded it by a constant factor times the largest term. Therefore, we can we can simplify the expression star to the following. I'm gonna write it in terms of big O notation. And, like, on the last slide, I'll use it to suppress two different constants. On the one hand, I'm gonna be suppressing the constant C, which we inherited way back when from the original recurrence. And on the other hand, I'm gonna use it to also suppress this constant that comes from the basic sums fact. So ignoring those two constants, what do we have left? We have N to the D. Times the largest term of the sum. So what is the largest term of the sum? Well, it's the last one so we plug in the biggest value of J that we're ever going to see. So what's the biggest value of J that we're ever going to see? We'll it's just this. Log base B of N. So, we get the ratio A over B to the D, raised to the log base B of N. Power. Now don't despair how messy this looks. We can do some remarkable simplifications. So what I want to do next is I want to focus just on this one over B to the D, raised to the log base B of N term. So that's going to be. You can write that as B to the minus D log base B of N. Which if I factor this exponent into two successive parts I can write this as B Raise to the log base B of N power. And only then raised to the minus D. And now of course what happens is that taking the logarithm of N base B, followed by taking, raising it to the B power, those are inverse operations that cancel, so that leaves us just with the N. So this results in a end to the minus D. And now remarkably this end to the minus D is just gonna cancel out with this end to the D. Leaving us with merely. A, raise the log based B event. And thus, out of this crazy sea of letters, rises a formula we can actually understand. So A to the log based B of N, if we step back and pick for a minute, is actually a supernatural quantity. Describe something about the recursion trees that we already knew was supposed to pop up in the analysis. I'll let, I'll let you think through exactly what that is in the following quiz. So the correct answer to this quiz is the fourth response. A raised to the logarithm event is precisely the number of leaves of the recursion tree. And remember in our intuition for case three, recursion trees where the amount of work is increasing per level, we thought that perhaps the work would be dominated by the work done at the leaves which is as proportional as the number of leaves. So why is this the answer? Well just remember what recursion trees look like at level zero. We have a single node, and then with each level we have eight times as many nodes as before. That is, with each level of the recursion tree, the number of nodes goes up by a factor of A. How far does this, how long does this process go on? Well, it goes on until we reach down the, the leaves. Recall that in the input size starts at N up at the root. It gets divided by a factor of B each time, and it terminates once we get down to one. So the leaves preside at precisely level log base B of N. So therefore. The number of leaves is just a branching factor which is A raised to the number of times that we actually multiply by A which is just the number of levels which is log base b n. So each time we go down a level we increase the number of nodes by a factor of A and we go down a level log base B of N times. Leaving us with a number of leaves equal to A raised to the log base B of N. So what we've done is we've mathematically confirmed, in a very cool way, our intuition about what case three should look like in the master method. We've proven that in case three when A is. Bigger than b to the d. The running time is, o of the number of leaves in the recursion tree, just as the intuition predicted. But, this leaves us with one final mystery. If you go back to the statement of the master method, we didn't say, a to the log base b of n. In case three, it says the running time is, n to the log base b of a. And, not only that, we've used this case three formula over and over again, to evaluate Gauss's recursive algorithm for integer multiplication, to evaluate the Strassen's matrix multiplication algorithm, and so on. So, what's the story? How come we're not getting the same thing, as in the statement of the master method? Well there's a very simple explanation, which is simply that, believe it or not. A log base B of N, and N to the log base B of A. Are exactly the same thing. This looks like the kind of mistake you'd make in freshmen algebra. But actually, if you think about it, these are simply the same quantity. If you don't believe me, just take the logarithm base B of both sides, and it'll give the same thing in both sides. Now, you might well be wondering why I didn't just state in the master method that the running time of case three is this very sensible and meaningful expression, a raised log based b of n, i.e., the number of leaves in the recursion tree. Well, it turns out that while this expression on the left hand side is the more meaningful conceptually. The right hand side. N. To the log base B. Of A. Is the easiest one to apply. So recall when we worked through a bunch of examples, of the master method, this right hand side was super convenient, when we evaluated the running times of out rhythms. When we plugged in the numbers of A. And B. In any case, whether or not you want to think about the running time in case three as proportional to the number of leaves in the tree or as proportional at the end of the log base B of A, we're done. We've proved it. That's case three. That was the last one. So we're done with the master method. Qed. So that was a lot of hard work for doing the master method and I would never expect someone to be able to regurgitate all of the details of this proof you know it's something like a cocktail party well maybe except the nerdiest of all cocktail parties but I do think there's a few high level conceptual points of this proof that are worth remembering in the long term, so we started by just writing down a recursion tree for the recursive algorithm and in a generic way. And going level by level, we counted up the work done by the algorithm. And this part of the proof had nothing to with how A and B related to each other. Then we recognized that there are three fundamentally different types of recursion trees. Those with the same amount of work per level, those where it increases with the level, and those where it decreases with the level. If you can remember that, you can even remember what the running times of the three cases should be. In the case where you do the same amount of every work at each level. We know there's a logarithmic number of levels. We know we do end in D work at the root. So that gives us the running time in case one had ended the day you log in. When the amount of work is decreasing with the levels, we now know that the route dominates. Up to a constant, we can throw out the rest of the levels, and we know end of the D work gets done at the root, so that's the overall running time. And in the third case, where it's increasing in the levels, the leaves dominate. The number of leaves is A raised to the log based of B of N, and that's the same as N, the log based B of A. And that's proportional to running time in case three of the master method.