wk3

# 0wk3-Overview

QUICKSORT - THE ALGORITHM (Part V). One of the greatest algorithms ever, and our first example of a randomized algorithm. These lectures go over the pseudocode --- the high-level approach, how to partition an array around a pivot element in linear time with minimal extra storage, and the ramifications of different pivot choices --- and explain how the algorithm works.

QUICKSORT - THE ANALYSIS (Part VI). These lectures prove that randomized QuickSort (i.e., with random pivot choices) runs in O(n log n) time on average. The analysis is as elegant as the algorithm itself, and is based on a "decomposition principle" that is often useful in the analysis of randomized algorithms. Note that there are some accompanying lecture notes for this part (available for download underneath each video). Also, it may be helpful to watch the first probability review video (below) before watching this sequence.

PROBABILITY REVIEW (Part VII). This first of these videos reviews the concepts from discrete probability that are necessary for the QuickSort analysis --- sample spaces, events, random variables, expectation, and linearity of expectation. The second video covers just two topics, although quite tricky ones! (Namely, conditional probability and independence.) You need to review these two topics (via this video or some other source, as you wish) before studying the analysis of the randomized contraction algorithm in Week 4.

HOMEWORK: This week's quiz will help you understand QuickSort and probability more deeply. Programming Assignment #3 asks you to implement QuickSort and compute the number of comparisons that it makes for three different pivot rules.

SUGGESTED READINGS FOR WEEK 3:

CLRS Chapter 7

KT Section 13.5

SW Section 2.3

# 0wk3-menu

V. QUICKSORT - ALGORITHM

Week 3 Overview10 分

Quicksort: Overview12 分

继续

Partitioning Around a Pivot24 分

Correctness of Quicksort [Review - Optional]10 分

Choosing a Good Pivot22 分

VI. QUICKSORT - ANALYSIS

Analysis I: A Decomposition Principle21 分

Analysis II: The Key Insight11 分

Analysis III: Final Calculations8 分

VII. PROBABILITY REVIEW

Probability Review I25 分

Probability Review II17 分

Problem Set #3

购买课程以解锁此项目。

测验: Problem Set #35 个试题

截止时间 六月 11, 11:59 晚上 PDT

Programming Assignment #3

购买课程以解锁此项目。

测验: Programming Assignment #33 个试题

截止时间 六月 11, 11:59 晚上 PDT

# 5-1-Quicksort\_Overview

So now we come to one of my favorite sequence of lectures, where we going to discuss the famous QuickSort algorithm. If you ask professional computer scientists and professional programmers to draw up a list of their top five, top ten favorite algorithms, I'll bet you'd see QuickSort on many of those, those peoples' lists. So, why is that? After all, we've already discussed sorting. We already have a quite good and practical sorting algorithm, mainly the Merge Sort algorithm. Well, QuickSort, in addition to being very practical, it's competitive with, and often superior to, Merge Sort. So, in addition to being very practical, and used all the time in the real world, and in programming libraries, it's just a extremely elegant algorithm. When you see the code, it's just so succinct. It's so elegant, you just sorta wish you had come up with it yourself. Moreover, the mathematical analysis which explains why QuickSort runs so fast, and that mathematical analysis, we'll cover in detail, is very slick. So it's something I can cover in just about half an hour or so. So more precisely what we'll prove about the QuickSort algorithm is that a suitable randomized implementation runs in time N log N on average. And I'll tell you exactly what I mean by on average, later on in this sequence of lectures. And, moreover, the constants hidden in the Big-Oh notation are extremely small. And, that'll be evident from the analysis that we do. Finally, and this is one thing that differentiates QuickSort from the merge sort algorithm, is it operates in place. That is, it needs very little additional storage, beyond what's given in the input array, in order to accomplish the goal of sorting. Essentially, what QuickSort does is just repeated swaps within the space of the input array, until it finally concludes with a sorted version of the given array. The final thing I want to mention on this first slide is that, unlike most of the videos, this set of the videos will actually have an accompanying set of lecture notes, which I've posted on, in PDF, from the course website. Those are largely, redundant. They're optional, but if you want another treatment of what I'm gonna discuss, a written treatment, I encourage you to look at the lecture notes, on the course website. So, for the rest of this video, I'm gonna give you an overview of the ingredients of QuickSort, and what we have to discuss in more detail, and the rest of the lectures will give details of the implementation, as well as the mathematical analysis. So let's begin by recalling the sorting problem. This is exactly the same problem we discussed back when we covered Merge Sort. So we're given as input an array of n numbers in arbitrary order. So, for example, perhaps the input looks like this array here. And then what do we gotta do? We just gotta output a version of these same numbers but in increasing order. Like when we discussed Merge Sort, I'm gonna make a simplifying assumption just to keep the lectures as simple as possible. Namely I'm going to assume the input array has no duplicates. That is, all of the entries are distinct. And like with the merge sort, I encourage you to think about how you would alter the implementation of QuickSort so that it deals correctly with ties, with duplicate entries. To discuss how QuickSort works at a high-level, I need to introduce you to the key subroutine, and this is really the, key great idea in QuickSort, which is to use a subroutine which partitions the array around a pivot element. So what does this mean? Well, the first thing you gotta do is, you gotta pick one element in your array to act as a pivot element. Now eventually we'll worry quite a bit about exactly how we choose this magical pivot element. But for now you can just think of it that we pluck out the very first element in the array to act as the pivot. So, for example, in the input array that I mentioned on the previous slide, we could just use "3" as the pivot element. After you've chosen a pivot element, you then re-arrange the array, and re-arrange it so that every, all the elements which come to the left of the pivot element are less than the pivot, and all the elements which come after the pivot element are greater than the pivot. So for example, given this input array, one legitimate way to rearrange it, so that this holds, is the following. Perhaps in the first two entries, we have the 2 and the 1. Then comes the pivot element. And then comes the elements 4 through 8 in some perhaps jingled order. So notice that the elements to the left of the pivot, the 2 and the 1, are indeed less than the pivot, which is 3. And the five elements to the right of the pivot, to the right of the 3, are indeed all greater than 3. Notice in the Partition subroutine, we do not insist that we get the relative order correct amongst those elements less than the pivot, or amongst those elements bigger than the pivot. So, in some sense, we're doing some kind of partial sorting. We're just bucketing the elements of the array into one bucket, those less than the pivot, and then a second bucket, those bigger than the pivot. And we don't care about, getting right the order amongst each, within each of those two buckets. So, partitioning is certainly a more modest goal than sorting, but it does make progress toward sorting. In particular, the pivot element itself winds up in its rightful position. That is, the pivot element winds up where it should be in the final sorted version of the array. You'll notice in the example, we chose as the pivot the third largest element, and it does, indeed, wind up in the third position of the array. So, more generally, where should the pivot be in the final sorted version? Well, it should be to the right of everything less than it. It should be to the left of everything bigger than it. And that's exactly what partitioning does, by definition. So, why is it such a good idea to have a partitioning subroutine? After all, we don't really care about partitioning. What we want to do is sort. Well, the point is that partitioning can be done quickly. It can be done in linear time. And it's a way of making progress toward having a sorted version of an array. And it's gonna enable a divide-and-conquer approach toward sorting the input array. So, in a little bit more detail, let me tell you about two cool facts about the Partition subroutine. I'm not gonna give you the code for partitioning here. I'm gonna give it to you on the next video. But, here are the two salient properties of the Partition subroutine, discussed in detail in the next video. So the first cool fact is that it can be implemented in linear, that, is O(N) time, where N is the size of the input array, and moreover, not just linear time but linear time with essentially no extra overhead. So we're gonna get a linear time of mutation, where all you do is repeated swaps. You do not allocate any additional memory. And that's key to the practical performance of the QuickSort algorithm. [sound] Secondly, it cuts down the problem size, so it enables the divide-and-conquer approach. Namely, after we've partitioned an array around some pivot elements, all we have to do is recursively sort the elements that lie on the left of the pivot. And recursively sort the elements that lie on the right of the pivot. And then, we'll be done. So, that leads us to the high-level description of the QuickSort algorithm. Before I give the high-level description, I should mention that this, algorithm was discovered by, Tony Hoare, roughly, 1961 or so. This was at the very beginning of Hoare's career. He was just about 26, 27 years old. He went on to do a lot of other contributions, and, eventually wound up winning the highest honor in computer science, the ACM Turing Award, in 1980. And when you see this code, I'll bet you feel like you wish you had come up with this yourself. It's hard not to be envious of the inventor of this very elegant QuickSort algorithm. So, just like in Merge Sort, this is gonna be a divide-and-conquer algorithm. So it takes an array of some length N, and if it's an array of length N, it's already sorted, and that's the base case and we can return. Otherwise we're gonna have two recursive calls. The big difference from Merge Sort is that, whereas in Merge Sort, we first split the array into pieces, recourse, and then combine the results, here, the recursive calls come last. So, the first thing we're going to do is choose a pivot element, then partition the array around that pivot element, and then do two recursive calls. And then, we'll be done. There will be no combined step, no merge step. So in the general case, the first thing you do is choose a pivot element. For the moment I'm going to be loose, leave the ChoosePivot subroutine unimplemented. There's going to be an interesting discussion about exactly how you should do this. For now, you just do it in some way, that for somehow you come up with one pivot element. For example, a naive way would be to just choose the first element. Then you invoke the Partition subroutine that we'll discuss in the last couple slides. [sound]. So recall that the results in a version of the array in which the pivot element p is in its rightful position, everything to the left of p is less than p, everything to the right of the pivot is bigger than the pivot, and then all you have to do to finish up is recurse on both sides. So let's call the elements less than p the first part of the partitioned array, and the elements greater than p the second part of the recursive array. And now we just call QuickSort again to recursively sort the first part, and then the, recursively sort the second part. And that is it. That is the entire QuickSort algorithm at the high-level. This is one of the relatively rare recursive, divide- and-conquer algorithms that you're going to see, where you literally do no work after solving the sub-problems. There is no combine step, no merge step. Once you've partitioned, you just sort the two sides and you're done. So that's the high- level description of the QuickSort algorithm. Let me give you a quick tour of what the rest of the video's going to be about. So first of all I owe you details on this Partition subroutine. I promise you it can be implemented in linear time with no additional memory. So I'll show you an implementation of that on the next video. We'll have a short video that formally proves correctness of the QuickSort algorithm. I think most of you will kinda see intuitively why it's correct. So, that's a video you can skip if you'd want. But if you do want to see what a formal proof of correctness for a divide-and-conquer algorithm looks like, you might want to check out that video. Then, we'll be discussing exactly how the pivot is chosen. It turns out the running time of QuickSort depends on what pivot you choose. So, we're gonna have to think carefully about that. Then, we'll introduce randomized QuickSort, which is where you choose a pivot element uniformly at random from the given array, hoping that a random pivot is going to be pretty good, sufficiently often. And then we'll give the mathematical analysis in three parts. We'll prove that the QuickSort algorithm runs in N log N time, with small constants, on average, for a randomly chosen pivot. In the first analysis video, I'll introduce a general decomposition principle of how you take a complicated random variable, break it into indicator random variables, and use linearity of expectation to get a relatively simple analysis. That's something we'll use a couple more times in the course. For example, when we study hashing. Then, we'll discuss sort of the key insight behind the QuickSort analysis, which is about understanding the probability that a given pair of elements gets compared at some point in the algorithm. That'll be the second part. And then there's going to be some mathematical computations just to sort of tie everything together and that will give us the bound the QuickSort running time. Another video that's available is a review of some basic probability concepts for those of you that are rusty, and they will be using in the analysis of QuickSort. Okay? So that's it for the overview, let's move on to the details.

# 5-2-Partitioning Around a Pivot

The goal of this video is to provide more details about the implementation of the QuickSort algorithm and, in particular, if you're ever going to drill down on the key Partition subroutine, just let me remind you what the job of the Partition subroutine is in the context of sorting an array. So recall that key idea in QuickSort is to partition the input array around a pivot element. So this has two steps. First, you somehow choose a pivot element, and in this video, we're not going to worry about how you choose the pivot element. For concreteness, you might just want to think about you pick the first element in the array to serve as your pivot. So in this example array, the first element happens to be 3, so we can choose 3 as the pivot element. Now, there's a key rearrangement step. So you rearrange the array so that it has the following properties. Any entries that are to the left of the pivot element should be less than the pivot element. Whereas any entries, which are to the right of the pivot element, should be greater than the pivot element. So, for example, in this, version of, the second version of the array, we see to the left of the 3 is the 2 and the 1. They're in reverse order, but that's okay. Both the 2 and the 1 are to the left of the 3, and they're both less than 3. And the five elements to the right of the 3, they're jumbled up, but they're all bigger than the pivot element. So, this is a legitimate rearrangement that satisfies the partitioning property. And, again, recall that this definitely makes partial progress toward having a sorted array. The pivot element winds up in its rightful position. It winds up where it's supposed to be in the final sorted array, to the right of everything less than it, to the left of everything bigger than it. Moreover, we've correctly bucketed the other N-1 elements to the left and to the right of the pivot according to where they should wind up in the final sorted array. So that's the job, that the Partition subroutine is responsible for. Now what's cool is we'll be able to implement this Partition subroutine in linear time. Even better, we'll be able to implement it so that all it does, really, is swaps in the array. That is, it works in-place. It needs no additional, essentially constant additional memory, to rearrange the array according to those properties. And then, as we saw on the high-level description of the QuickSort algorithm, what partitioning does is, it enables a divide-and-conquer approach. It reduces the problem size. After you've partitioned the array around the pivot, all you gotta do is recurse on the left side, recurse on the right side, and you're done. So, what I owe you is this implementation. How do you actually satisfy the partitioning property, stuff to the left of the pivot is smaller than it, stuff to the right of the pivot is bigger than it, in linear time, and in- place. Well, first, let's observe that, if we didn't care about the in-place requirement, if we were happy to just allocate a second array and copy stuff over, it would actually be pretty easy to implement a Partition subroutine in linear time. That is, using O(N) extra memory, it's easy to partition around a pivot element in O(N) time. And as usual, you know, probably I should be more precise and write theta of N, are used in cases that would be the more accurate stronger statement, but I'm going to be sloppy and I'm just going to write the weaker but still correct statement, using Big-Oh, okay? So O(N) time using linear extra memory. So how would you do this? Well let me just sort of illustrate by example. I think you'll get the idea. So let's go back to our running example of an input array. Well, if we're allowed to use linear extra space, we can just preallocate another array of length N. Then we can just do a simple scan through the input array, bucketing elements according to whether they are bigger than or less than the pivot. And, so for example, we can fill in the additional array both from the left and the right, using elements that are less than or bigger than the pivot respectively. So for example we start with the 8, we know that the 8 is bigger than the pivot, so you put that at the end of the output array. Then we get to the 2. The 2 is less than the pivot, so that should go on the left hand side of the output array. When you get to the 5, it should go on the right-hand side, and the 1 should go on the left-hand side, and so on. When we complete our scan through the input array, there'll be one hole left, and that's exactly where the pivot belongs, to the right of everything less than it, to the left of everything bigger than it. So, what's really interesting, then, is to have an implementation of Partition, which is not merely linear time, but also uses essentially no additional space. It doesn't re-sort to this cop-out of pre-allocating an extra array of length N. So, let's turn to how that works. First, starting at a high-level, then filling in the details. So I'm gonna describe the Partition subroutine only for the case where the pivot is in fact the first element. But really this is without loss of generality. If, instead, you want to use some pivot from the middle of the array, you can just have a preprocessing step that swaps the first element of the array with the given pivot, and then run the subroutine that I'm about to describe, okay. So with constant time preprocessing, the case of a general pivot reduces to the case of when the pivot is the first element. So here's the high-level idea, and it's very cool. The idea is, we're gonna be able to able to get away with just a single linear scan of the input array. So in any given moment in this scan, there's just gonna be a single for-Loop, we'll be keeping track of both the part of the array we've looked at so far, and the part that we haven't looked at so far. So there's gonna be two groups, what we've seen, what we haven't seen. Then within the group we've seen, we're gonna have definitely split further, according to the elements that are less than the pivot and those that are bigger than the pivot. So we're gonna leave the pivot element just hanging out in the first element of the array until the very end of the algorithm, when we correct its position with a swap. And at any given snapshot of this algorithm, we will have some stuff that we've already looked at, and some stuff that we haven't yet looked at in our linear scan. Of course, we have no idea what's up with the elements that we haven't looked at yet, who knows what they are, and whether they're bigger or less than the pivot. But, we're gonna implement the algorithm, so, among the stuff that we've already seen, it will be partitioned, in the sense that all elements less than the pivot come first, all elements bigger than the pivot come last. And, as usual, we don't care about the relative order, amongst elements less than the pivot, or amongst elements bigger than the pivot. So summarizing, we do a single scan through the input array. And the trick will be to maintain the following invariant throughout the linear scan. But basically, everything we have looked at the input array is partitioned. Everything less than the pivot comes before everything bigger than the pivot. And, we wanna maintain that invariant, doing only constant work, and no additional storage, with each step of our linear scan. So, here's what I'm gonna do next. I'm gonna go through an example, and execute the Partition subroutine on a concrete array, the same input array we've been using as an example, thus far. Now, maybe it seems weird to give an example before I've actually given you the algorithm, before I've given you the code. But, doing it this way, I think you'll see the gist of what's going on in the example, and then when I present the code, it'll be very clear what's going on. Whereas, if I presented the code first, it may seem a little opaque when I first show you the algorithm. So, let's start with an example. Throughout the example, we wanna keep in mind the high-level picture that we discussed in the previous slide. The goal is that, at any time in the Partition subroutine, we've got the pivot hanging out in the first entry. Then, we've got stuff that we haven't looked at. So, of course, who knows whether those elements are bigger than or less than the pivot? And then, for the stuff we've looked at so far, everything less than the pivot comes before everything bigger than the pivot. This is the picture we wanna retain, as we go through the linear scan. As this high-level picture would suggest, there is two boundaries that we're gonna need to keep track of throughout the algorithm. We're gonna need to keep track of the boundary between what we've looked at so far, and what we haven't looked at yet. So, that's going to be, we're going to use the index "j" to keep track of that boundary. And then, we also need a second boundary, for amongst the stuff that we've seen, where is the split between those less than the pivot and those bigger than the pivot. So, that's gonna be "i". So, let's use our running example array. >> So stuff is pretty simple when we're starting out. We haven't looked at anything. So all of this stuff is unpartitioned. And "i" and "j" both point to the boundary between the pivot and all the stuff that we haven't seen yet. Now to get a running time reaches linear, we want to make sure that at each step we advance "j", we look at one new element. That way in a linear number of steps, we'll have looked at everything, and hopefully we'll be done, and we'll have a partitioned array. So, in the next step, we're going to advance "j". So the region of the array which is, which we haven't looked at, which is unpartitioned, is one smaller than before. We've now looked at the 8, the first element after the pivot. Now the 8 itself is indeed a partitioned array. Everything less than the pivot comes before, everything after the pivot turns out there's nothing less than the pivot. So vacuously this is indeed partitioned. So "j" records delineates the boundary between what we've looked at and what we haven't looked at, "i" delineates amongst the stuff we've looked at, where is the boundary between what's bigger than and what's less than the pivot. So the 8 is bigger than the pivot, so "i" should be right here. Okay, because we want "i" to be just to the left of all the stuff bigger than the pivot. Now, what's gonna happen in the next iteration? This is where things get interesting. Suppose we advance "j" one further. Now the part of the array that we've seen is an 8 followed by a 2. Now an 8 and a 2 is not a partitioned subarray. Remember what it means to be a partitioned subarray? All the stuff less than the pivot, all the stuff less than 3, should come before everything bigger than 3. So (8, 2) obviously fails that property. 2 is less than the pivot, but it comes after the 8, which is bigger than the pivot. So, to correct this, we're going to need to do a swap. We're going to swap the 2 and the 8. That gives us the following version of the original array. So now the stuff that we have not yet looked at is one smaller than before. We've advanced "j". So all other stuff is unpartitioned. Who knows what's going on with that stuff? "j" is one further entry to the right than it was before, and at least after we have done this swap, we do indeed have a partitioned array. So post-swap, the 2 and the 8, are indeed partitioned. Now remember, "I" delineates the boundary between amongst what we've seen so far, the stuff less than the pivot, less than 3 in this case, and that bigger than 3, so "I" is going to be wedged in between the 2 and the 8. In the next iteration, our life is pretty easy. So, in this case, in advancing "j", we uncover an element which is bigger than the pivot. So, this is what happened in the first iteration, when we uncovered the 8. It's different than what happened in the last iteration when we uncovered the 2. And so, this case, this third iteration is gonna be more similar to the first iteration than the second iteration. In particular, we won't need to swap. We won't need to advance "i". We just advance "j", and we're done. So, let's see why that's true. So, we've advanced "j". We've done one more iteration. So, now the stuff we haven't seen yet is only the last four elements. So, who knows what's up with, the stuff we haven't seen yet? But if you look at the stuff we have seen, the 2, the 8, and the 5, this is, in fact, partitioned, right? All the numbers that are bigger than 3 succeed, come after, all the numbers smaller than three. So the "j", the boundary between what we've seen and what we haven't is between the 5 and the 1; and the "i", the boundary between the stuff less than the pivot and bigger than the pivot is between the 2 and the 8, just like it was before. Adding a 5 to the end didn't change anything. So let's wrap up this example in the next slide. So first, let's just remember where we left off from the previous slide. So I'm just gonna redraw that same step after three iterations of the algorithm. And notice, in the next generation, we're going to, again, have to make some modifications to the array, if we want preserve our variant. The reason is that when we advance "j", when we scan this 1, now again we're scanning in a new element which is less than the pivot, and what that means is that, the partitioned region, or the region that we've looked at so far, will not be partitioned. We'll have 2851. Remember we need everything less than 3 to precede everything bigger than 3, and this 1 at end is not going to cut it. So we're going to have to make a swap. Now what are we going to swap? We're going to swap the 1 and the 8. So, why do we swap the 1 and the 8? Well, clearly, we have to swap the 1 with something. And, what makes sense? What makes sense is the left-most array entry, which is currently bigger than the pivot. And, that's exactly the 8. Okay, that's the first, left-most entry bigger than 3, so if we swap the 1 with it, then the 1 will become the right-most entry smaller than 3. So after the swap, we're gonna have the following array. The stuff we haven't seen is the 4, the 7, and the 6. So the "j" will be between the 8 and the 4. The stuff we have seen is the 2, 1, 5, and 8. And notice, that this is indeed partitioned. All the elements, which are less than 3, the 2 and the 1, precede all of the entries, which are bigger than 3, the 5 and the 8. "i", remember, is supposed to split, be the boundary between those less than 3 and those bigger than 3. So, that's gonna lie between the 1 and the 5. That is one further to the right than it was in the previous iteration. Okay, so the, because the rest of the unseen elements, the 4, the 7, and the 6, are all bigger than the pivot, the last three iterations are easy. No further swaps are necessary. No increments to "i" are necessary. "j" is just going to get incremented until we fall off the array. And then, fast forwarding, the Partition subroutine, or this main linear scan, will terminate with the following situation. So at this point, all of the elements have been seen, all the elements are partitioned. "j" in effect has fallen off the end of the array, and "i", the boundary between those less than and bigger than the pivot, still lies between the 1 and the 5. Now, we're not quite done, because the pivot element 3 is not in the correct place. Remember, what we're aiming for is an array where everything less than the pivot is to the left of it, and everything bigger than the pivot is to the right. But right now, the pivot still is hanging out in the first element. So, we just have to swap that into the correct place. Where's the correct place? Well, it's going to be the right-most element, which is smaller than the pivot. So, in this case, the 1. So the subroutine will terminate with the following array, 12358476. And, indeed, as desired, everything to the left of the pivot is less than the pivot, and everything to the right of the pivot is bigger than the pivot. The 1 and 2 happen to be in sorted order, but that was just sorta an accident. And the 4, 5, 6 and 7 and 8, you'll notice, are jumbled up. They're not in sorted order. So hopefully from this example you have a gist of how the Partition subroutine is going to work in general. But, just to make sure the details are clear, let me now describe the pseudocode for the Partition subroutine. So the way I'm going to denote it is, there's going to be an input array A. But rather than being told some explicit link, what's going to be passed to the subroutine are two array indices. The leftmost index, which delineates this part of the separator you're supposed to work on, and the rightmost index. The reason I'm writing it this way is because Partition is going to be called recursively from within a QuickSort algorithm. So any point in QuickSort, we're going to be recursing on some subset, contiguous subset of the original input array. "l(el)" and "r" meant to denote what the left boundary and the right boundary of that subarray are. So, let's not lose sight of the high-level picture of the invariant that the algorithm is meant to maintain. So, as we discussed, we're assuming the pivot element is the first element, although that's really without loss of generality. At any given time, there's gonna be stuff we haven't seen yet. Who knows what's up with that? And, amongst the stuff we've seen, we're gonna maintain the invariant that all the stuff less than the pivot comes before all the stuff bigger than the pivot. And "j" and I denote the boundaries, between the seen and the unseen, and between the small elements and the large elements, respectively. So back to the pseudocode, we initialize the pivot to be the first entry in the array. And again remember, l denotes the leftmost index that we're responsible for looking at. Initial value of "i", should be just to the right of the pivot so that's gonna be el+1. That's also the initial value of "j", which will be assigned in the main for-Loop. So this for-Loop with "j", taking on all values from el+1 to the rightmost index "r", denotes the linear scan through the input array. And, what we saw in the example is that there were two cases, depending on, for the newly seen element, whether it's bigger than the pivot, or less than the pivot. The easy case is when it's bigger than the pivot. Then we essentially don't have to do anything. Remember, we didn't do any swaps, we didn't change "i", the boundary didn't change. It was when the new element was less than the pivot that we had to do some work. So, we're gonna check that, is the newly seen element, A[j], less than "p". And if it's not, we actually don't have to do anything. So let me just put as a comment. If the new element is bigger than the pivot, we do nothing. Of course at the end of the for-Loop, the value of "j" will get in command so that's the only thing that changes from iteration to iteration, when we're sucking up new elements that happen to be bigger than "p". So what do we do in the example, when we suck up our new element less than p? Well we have to do two things. So, in the event that the newly seen element is less than "p", I'll circle that here in pink. We need to do a rearrangement, so we, again, have a partitioned, sub-array amongst those elements we've seen so far. And, the best way to do that is to swap this new element with the left-most element that's bigger than the pivot. And because we have an index "i", which is keeping track of the boundary between the elements less than the pivot and bigger than the pivot, we can immediately access the leftmost element bigger than the pivot. That's just the "i"th entry in the array. Now I am doing something a little sneaky here, I should be honest about. Which is there is the case where you haven't yet seen any elements bigger than the pivot, and then you don't actually have a leftmost element bigger than the pivot to swap with. Turns out this code still works, I'll let you verify that, but it does do some redundant swaps. Really, you don't need to do any swaps until you first see some elements bigger than the pivot, and then see some elements less than the pivot. So, you can imagine a different limitation of this, where you actually keep track of whether or not that's happened to avoid the redundant swaps. I'm just gonna give you the simple pseudocode. And again, for intuition, you wanna think about the case just like, in the picture here in blue, where we've already seen some elements that are bigger than the pivot, and the next newly seen element is less than the pivot. That's really sort of the key case here. Now the other thing we have to do after one of these swaps is, now the boundary, between where the array elements less than the pivot and those bigger than the pivot, has moved. It's moved one to the right, so we have to increment "i". So, that's the main linear scan. Once this concludes, "j" will have fallen off the end of the array. And, everything that we've seen the final elements, except for the pivot, will be arranged so that those less than "p" are first, those bigger than "p" will be last. The final thing we have to do is just swap the pivot into its rightful position. And, recall for that, we just swap it with the right-most element less than it. So, that is it. That is the Partition subroutine. There's a number of variants of partition. This is certainly not the unique implementation. If you look on the web, or if you look in certain textbooks, you'll find some other implementations as well as discussion of the various merits. But, I hope this gives you, I mean, this is a canonical implementation, and I hope it gives you a clear picture of how you rearrange the array using in-place swaps to get the desired property, that all the stuff before the pivot comes first, all the stuff after the pivot comes last. Let me just add a few details about why this pseudocode I just gave you does, indeed, have the properties required. The running time is O(N), really theta of N, but again, I'll be sloppy and write O(N). Where N is the number of array elements that we have to look at. So, N is r-el+1, which is the length of the sub-array that this Partition subroutine is invoked upon. And why is this true? Well if you just go inspect the pseudocode, you can just count it up naively and you'll find that this is true. We just do a linear scan through the array and all we do is basically a comparison and possibly a swap and an increment for each array entry that we see. Also, if you inspect the code, it is evident that it works in-place. We do not allocate some second copy of an array to populate, like we did in the naive Partition subroutine. All we do is repeated swaps. Correctness of the subroutine follows by induction, so in particular the best way to argue it is by invariant. So I'll state the invariant here, but mostly leave it for you to check that indeed, every iteration of the for-Loop maintains this invariant. So first of all, all of the stuff to the right of the pivot element, to the right of the leftmost entry and up to the index "i", is indeed less than the pivot element, as suggested by the picture. And also suggested by the picture, everything beginning with the "i"th entry, leading just up before the "j"th entry, is bigger than the pivot. And I'll leave it as a good exercise for you to check that this holds by induction. The invariant holds initially, when both "i" and "j" are equal to el+1, because both of these sets are vacuous, okay? So, there are no such elements, so they're trivially satisfied these properties. And then, every time we advance "j", well, in one case it's very easy, where the new element is bigger than the pivot. It's clear that, if the invariant held before, it also holds at, at the next iteration. And then, if you think about it carefully, this swap in this increment of "i" that we do, in the case where the new element is less than the pivot. After the swap, once the fold is complete, again if this invariant was true at the beginning of it, it's also true at the end. So what good is that? Well, by this claim, at the conclusion of the linear scan at which point "j" has fallen off the end of the array, the array must look like this. At the end of the for-Loop, the question mark part of the array has vanished, so everything other than the pivot has been organized so that all this stuff less than the pivot comes before everything after the pivot, and that means once you do the final swap, once you swap the pivot element from its first and left most entry, with the right most entry less than the pivot, you're done. Okay? You've got the desired property that everything to the left of the pivot is less than, and everything to the right of the pivot is bigger than. So now that given a pivot element we understand how to very quickly rearrange the array so that it's partitioned around that pivot element, let's move on to understanding how that pivot element should be chosen and how, given suitable choices of that pivot element, we can implement the QuickSort algorithm, to run very quickly, in particular, on average in O(N) log time.

# 5-3-Correctness of Quicksort [Review - Optional]

I just got the number of divide and conquer algorithms and, so far, I've been short shrift to proofs of correctness. This has been a conscience decision on my part. Coming up with the right divide and conquer algorithm for a problem can definitely be difficult, but once you have that eureka moment and you figure out the right algorithm you tend to, also, have a good understanding of why it's correct, why it actually solves the problem on every possible input. Similarly when I present to you a divide and conquer algorithm like, say, merge sort or quicksort, I expect that many of you have a good and accurate intuition about why the algorithm is correct. In contrast the running time of these developed [inaudible] algorithms is often highly non-obvious. So, correctness proofs for divide-and-conquer algorithms tend to simply formalize the intuition that you have via a proof by induction. That's why I haven't been spending much time on them. But nevertheless, I do feel like I owe you at least one rigorous correctness proof for a divide-and-conquer algorithm, and we may as well do it for quicksort. So in this optional video, we'll briefly review proofs by induction, and then we'll show how such a proof can be used to rigorously establish the correctness of quicksort. The correctness proofs for most of the other divide-and-conquer algorithms that we discuss can be formalized in a similar way. So let's begin by reviewing the format for proofs by induction. So, the canonical proofs by induction and the kind that we'll be using here, is when you want to establish an assertion for all of the positive integers in. So now it's some assertion which is parameterized by n, where n is a positive integer. I know this is a little abstract, so let me just be concrete about the assertion that we actually care about for quicksort. So for us, the assertion P(n) is the statement that cor, quicksort is always correct on inputs of length n, arrays that have n elements. So an induction proof has two parts. The first part is a base case and the second part is an inductive step. For the base case you have to get started so you show that at the very least your assertion is true when n equals one. This is often a trivial matter and that'll be the case when we establish the correctness of quick sort. Just on our rays with only one element. So, the non-trivial part of a proof by induction is usually the inductive step. And in the inductive step, you look at a value of n not covered by the base case, so a value of n bigger than one. And you show that if the assertion holds for all smaller values, small integers, then it also holds for the integer n. That is, you show that for every positive integer N that's two or greater, you assume that P of K holds for all K strictly less than N. And under that assumption, which is called the inductive hypothesis. Under the assumption that P of K holds for all K strictly less than N, you then establish that P of N holds as well. So if you manage to complete both of these steps, if you prove both the base case that P(1) holds, you argue that directly, and then also you argue that assuming the inductive hypothesis, that the assertion holds for all smaller integers, it also holds for an arbitrary integer n. Then you're done. Then in fact you have proven that the assertion P then holds for every single positive integer N. Right? So for any given N that you care about, the way you can derive that from one and two is you just start from the base case, P of one holds. Then you apply the inductive step N minus one times. And boom, you've got it. So you know that P holds for the integer N that you care about as well. And that's true for arbitrarily large values of N. So those are proofs by induction in general. Now let's instantiate this proof format, this type of proof for establishing the correctness of quicksort. So let me write again what is the assertion we care about. Our definition of P(n) is gonna be that quicksort is always correct on arrays of length n. And of course what we want to prove is that quicksort is correct no matter what size array that you give it, that is, we want to prove that P(n) holds for every single n at least one. So this is right in the wheelhouse of proofs by induction. ?Kay, so that's how we're going to establish it. Now depending on the order in which you're watching the videos, you may or may not have seen our discussion about how you actually choose the pivot, recall that the first thing Quick Sort does is choose a pivot, then it partitions the array around the pivot. So, we're going establish the correctness of Quick Sort, no matter how the choose pivot sub-routine gets implemented. Okay, so now matter how you choose pivots, you'll always have correctness. As we, as we'll see in a different video, the choice of pivot definitely has an influence on the running of Quick Sort, the correctness of Quick Sort, there's no matter how you choose the pivot. So it's perceived by a proof by induction. So for the base case when n equals one, this is a fairly trivial statement. Right? So, then we're just talking about inputs that have only one element. Every such array is already sorted. Quicksorts, in the bai, when n equals one just returns the input array. It doesn't do anything, and that is indeed the sort of array that it returns. So, by the rather trivial argument we had directly proven that p of one holds. We've proven the rather unimpressive statement that quicksort always correctly sorts one element arrays. Okay? No big deal. So, let's move on to the inductive step. So in the inductive step we have to fix an arbitrary value of N that's at least two. A value of N not covered by the base case. So let's fix some value of N, that leaves two. Now what are we trying to prove? We're trying to prove that Quick Sort always correctly sorts every input array of length N. So we also have to fix an arbitrary such input. So let's make sure we're all clear on what it is we need to show, what do you show in an inductive step. Assuming that PFK holds. For all smaller values, all smaller integers, then P of N holds as well. And remember this is the inductive hypothesis. So in the context of quicksort, we're assuming that quicksort never makes a mistake on any input array that has length strictly smaller than n. And now we just have to show it never makes a mistake on array, input arrays that have size exactly n. So this is the point in the proof where we actually delve into how Quick Sort is implemented to argue correctness. So recall what the first step of Quick Sort is, it picks some pivot arbitrarily, we don't know how, we don't care how. And then it partitions the array around this pivot element p. Now as we argued in the video where we discussed the partition sub routine, at the conclusion of that sub routine, the array has been rearranged into the following format. The pipit is wherever it is, everything to the left of the pipit is less than the pipit, and everything bigger than the pipit is greater than the pipit. Alright, this is where how things stand at the conclusion of the partitioning sub routine. So let's call this stuff less than the pipit the first part of the partition array, and the stuff bigger than the pipit, the second part of the partition array. And recall our observation from the overview video that the pivot winds up in its correct position. Right, where would the pivot be? Where is any element suppose to be in the final sorted array? What's suppose to be to the right of everything less than it, and to the left of everything bigger than it? And that's exactly where this partitioning subroutine deposits the pivot element peak. So now to imply the inductive hypothesis, which you'll recall is a hypothesis about how quick sort operates on smaller sub arrays. Let's call the length of the first part in the second part of the partition [inaudible] K1 and K2 respectively. Now, crucially, both k1 and k2 are strictly less than n. Both of these two parts have lengths strictly less than that of the given input array a. That's because the pivot in particular is excluded from both of those two parts. So, their gonna have, at most n minus one [inaudible]. That means that we can apply the inductive hypothesis, which says that the quicksort never makes a mistake on an array that has size strictly less than n. That implies that our two recursive calls to quickstart, the one to the first part and the one to the second part don't make mistakes. They're guaranteed to sort those sub arrays correctly by the inductive hypothesis. And to be very precise, what we're using to argue that the [inaudible] are correct, are P of K1 and P of K2. Or P is the assertion that [inaudible] is always correct on a [inaudible]. K1 and K2. And we know that both of these statements are true because k1 and k2 are less th, are both less than n and because of the inductive hypothesis. So what's the upshot? The upshot is, quicksort's gonna be correct. And so the first recursive call puts all of the elements that are less than the pivot in the correct relative order. Next comes the pivot, which is bigger than all of that stuff in the first part and less than all the stuff in the second part, and then the second recursive call correctly orders all of the elements in the second part. So with those three things pasted together, we have a sorted version of the input array and since this array was an arbitrary one, of link N. That establishes the assertion P of N and since n was arbitrary, that establishes the inductive and completes the proof of correctness of quick sort for an arbitrary method of choosing the pivot element. [sound]

# 5-4-Choosing a Good Pivot

So let's review the story so far. We've been discussing the QuickSort algorithm. Here again is its high level description. So in QuickSort you call two subroutines first, and then you make two recursive calls. So the first subroutine ChoosePivot, we haven't discussed yet at all. That'll be one of the main topics of this video. But the job of the ChoosePivot subroutine is to somehow select one of the n elements in the input array, to act as a pivot element. Now what does it mean to be a pivot? Well that comes into play in the second subroutine, the partition subroutine, which we did discuss quite a bit in a previous video. So what a partition does is it rearranges the elements in the input array, so that it has the following property, so that the pivot p winds up in its rightful position. That is, it's to the right of all of the elements less than it, and it's to the left of all of the elements bigger than it. The stuff less than it's to the left in some jumbled order. The stuff bigger than it's to the right in some jumbled order. That's what's listed here as the first part and the second part of the partitioned array. Now, once you've done this partitioning, you're good to go. You just recursively sort the first part to get them in the right order, you call QuickSort again to recursively sort the right part, and bingo, the entire array is sorted. You don't need a combine step, you don't need a merge step. Where we'll recall in a previous video, we saw that the partition array can be implemented in linear time. And moreover, it works in place with essentially no additional storage. We also, in an optional video, formally proved the correctness of QuickSort, and remember QuickSort is independent of how you implement the ChoosePivot subroutine. So what we're going to do now is discuss the running time of the QuickSort algorithm, and this is where the choice of the pivot is very important. So what everybody should be wondering about at this point is, is QuickSort a good algorithm? Does it run fast? The bar's pretty high. We already have MergeSort, which is a very excellent, practical n log n algorithm. The key point to realize at this juncture, is that we are not currently in a position to discuss the running time of the QuickSort algorithm. The reason is we do not have enough information. The running time of QuickSort depends crucially on how you choose the pivot. It depends crucially on the quality of the pivot chosen. You'd be right to wonder what I mean by a pivot's quality. And basically what I mean, is a pivot is good if it splits the partitioned array into roughly two equal sized subproblems. And it's bad, it's of low quality, if we get very unbalanced subproblems. So to understand both, what I mean, and the ramifications of having good quality and bad quality pivots, let's walk through a couple of quiz questions. This first quiz question is meant to explore a sort of worst case execution of the QuickSort algorithm. What happens when you choose pivots that are very poorly suited for the particular input array? Let me be more specific. Suppose we use the most naive ChoosePivot implementation, like we were discussing in the partition video. So remember, here we just pluck out the first element of the array and we use that as the pivot. So suppose that's how we implement the ChoosePivot subroutine, and moreover, suppose that the input array to QuickSort is an array that's already in sorted order. So for example, if it just had the numbers one through eight, it would be one, two, three, four, five, six, seven, eight, in order. My question for you is, what is the running time of this recursive QuickSort algorithm on an already sorted array, if we always use the first element of a subarray as the pivot? Okay, so this is a slightly tricky, but actually a very important question. So the answer is the fourth one. So it turns out, that QuickSort, if you pass it an already sorted array and you're using the first element as pivot elements, it runs in quadratic time. And remember for a sorting algorithm, quadratic is bad. It's bad in the sense that we can do better. MergeSort runs in time n log n, which is much better than n squared. And if we we're happy with an n squared running time, we wouldn't have to resort to these sort of relatively exotic sorting algorithms. We could just use Insertion sort, and we'd be fine. We'd get that same quadratic running time. Okay, so now I owe you an explanation. Why is it that QuickSort can actually run in quadratic time, in this unlucky case, of being passed an already sorted input array? Well to understand, let's think about what pivot gets chosen, and what are the ramifications of that pivot choice for how the array gets partitioned, and then what the recursion looks like. So, let's just think of the array as being the numbers 1 through n, in sorted order. What is going to be our pivot? Well, by definition we're choosing the first element of the pivot, so the pivot's just going to be 1. Now we're going to invoke the partition subroutine. And if you go back to the Pseudocode of the partition subroutine, you'll notice that if we pass an already sorted array, it's going to do essentially nothing. Okay? So it's just going to advance the index j, until it falls off the end of the array, and it's just going to return back to us, the same array that it was passed as input. So partition subroutine, if given an already sorted array, returns an already sorted array. So we have just a pivot 1, in the first position. And then the numbers 2 through n, in order, in the remainder of the positions. So if we draw our usual picture of what a partitioned array looks like, with everything less than the pivot to the left, everything bigger than the pivot to the right. Well, since nothing is less than the pivot, this stuff is going to be empty. This will not exist. And to the right of the pivot, this will have length n- 1, and moreover, it will still be sorted. So once partition completes, we go back to the outer call of QuickSort, which then calls itself recursively twice. Now in this case, one of the recursive calls is just vacuous. There's just an empty array, there's nothing to do. So really there's only one recursive call, and that happens on a problem of size only one less. So this is about the most unbalanced split we could possibly see, right, where one side has 0 elements, one side's n- 1. Splits don't really get any worse than that. And this is going to keep happening over, and over, and over again. We're going to recurse on the numbers 2 through n. We're going to choose the first element, the 2, as the pivot. Again, we'll feed it to partition. We'll get back the exact same subarray that we handed it in. We get to the numbers 2 through n, in sorted order. We exclude the pivot 2, we recurse on the numbers 3 through n, a subarray of length n- 2. The next recursion level, we recurse on an array of size of length n- 3, then n- 4, then n- 5, and so on. Until finally, after I did recursion depth of n, roughly, we got down to just the last element n, the base case kicks in, and we return that, and QuickSort completes. So that's how QuickSort is going to execute on this particular input with these particular pivot choices, so what running time does that give to us? Well, the first observation is that in each recursive call, we do have to invoke the partition subroutine. And the partition subroutine does look at every element in the array it has passed as input. So if we pass partition in array of length k, it's going to do at least k operations, because it looks at each element at least once. So the runtime is going to be bounded below by the work we do in the outermost call, which is on an array of length n, plus the amount we do in the second level of recursion, which is on a subarray of length (n- 1) + (n- 2) +, blah, blah, blah, blah, blah, all the way down to + 1, for the very last level of the recursion. So this is a lower bound on our running time, and this is already Theta of n squared. So, one easy way to see why this sum n + (n- 1) +, etc., etc., leads to a bound of n squared, is to just focus on the first half of the terms. So, the first n over two terms in the sum are all of magnitude at least n over 2, so the sum is at least n squared over 4. It's also evident that this sum is at most, n squared. So, overall, the running time of QuickSort on this bad input is going to be quadratic. Now having understood what the worst case performance for the QuickSort algorithm is, lets move on to discuss it's best case running time. Now we don't generally care about the best case performance of algorithms for it's own sake. The reason that we want to think about QuickSort in the best case, first of all it'll give us better intuition for how the algorithm works. Second of all, it'll draw a line in the sand. Its average case running time certainly can't be better than the best case, so this will give us a target for what we're shooting for in our subsequent mathematical analysis. So what were the best case? What was the highest quality pivot we could hope for? Well again, we think of the quality of the pivot as the amount of balance that it provides between the two sub problems. So ideally, we choose a pivot which gave us two sub-problems, both of size n over 2 or less. And there's a name for the element that would give us that perfectly balanced split. It's the median element of the array, okay, the element where exactly half of the elements are less than it and half of the elements are bigger than it. That would give us an essentially perfect 50, 50 split of the input array. So, here's the question. Suppose we had some input and we ranked QuickSort, and everything just worked in our favor, in the magically, in the best possible way. That is, in every single recursive invocation of QuickSort, on any sub array of the original input array. Suppose, we happen to get, as our pivot the median element. That is, suppose in every single recursive call. We wind up getting a perfect 50/50 split of the input array before we recurse. This question asks you to analyze the running time of this algorithm in this magical best case scenario. So the answer to this question is the third option. It runs (n log n) times. Why is that? Well, the reason is then the recurrence which governs the running time of QuickSort exactly matches the recurrence short running time that we already know is n log n. That is the running time QuickSort requires in this magical special case on a array of length n. As usual, you have a recurrence in two parts. There's the work that gets done by the recursive cause and there's the work that gets done now. Now by assumption, we wind up picking the median as the pivot. So there's going to be two recursive calls, each of which will be on an input of size at most N over two. And, we can write this, this is because the pivot equals the median. So this is not true for quick sort of general, it's only true in this magical case, where the pivot is the median. So that's what gets done by the two recursive calls. And then how much work do we do outside of the recursive calls? Well, we have to do the truth pivot subroutine. And I guess, strictly speaking, I haven't said how that was implemented. But let's assume that choose pivot does only a linear amount of work. And then, as we've seen, the partition subroutine only does a linear amount of work, as well. So let's say O(n), for work outside of the recursive calls. And what do we know? We know this implies, say by using the master method, or just by using the exact same argument as for Merge Sort, this gives us a running time balunt of (nlogn) And again something I haven't really been emphasizing but which is true is that actually we can write theta (n log n). And that's because in the recurrence, in fact, we know that the work done outside of the recursive calls is exactly theta (n), okay? Partition needs really linear time, not just O(n) time. In fact the work done outside of the recursive calls is theta (n). That's because the partition serpentine does indeed look at every entry in the array that it passed, all right, and as a result, we didn't really discuss this so much in the master method. But as I mentioned in passing, if you have recurrences which are tight in this sense, then the result of the master method can also be strengthened to be theta instead of just beta. But those are just some extra details. The upshot of this quiz is that even in the best case, even if we magically get prefect pivots throughout the entire trajectory of quick sort. The best we can hope for is an n log n upper bound, it's not going to get any better than that. So the question is how do we have a principled way of choosing pivots so that we get this best case or something like it that's best case n log n running time. So that's what the problem that we have to solve next. So the last couple quizzes have identified a super important question, as far as the implementation of Quicksort, which is how are we going to choose these pivots? We now know that they have a big influence on the running time of our algorithm. It could be as bad as n squared or as good as m log n, and we really want to be on the m log n side. So the key question: how to choose pivots. And quick sort is the first killer application that we're going to see of the idea of randomized algorithms, that is allowing your algorithms to flip coins in the code so that you get some kind of good performance on average. So the big idea is random pivots. By which I mean for every time we recursively call quick sort and we are pass some subarray of length k. Among the K candidate pivot elements in the sub-array, we're going to choose each one with probability of one over k. And we're going to make a new random choice every time we have qa recursive call, and we're going to see how the algorithm does. This is our first example of a randomized algorithm. This is an algorithm where, if you feed it exactly the same input, it'll actually run differently, on different execution. And that's because there's randomness internal to the code of the algorithm. Now, it's not necessarily intuitive. The randomization should have any purpose in the computation, in software design and algorithm design. But, in fact, and this has been sort of one of the real breakthroughs in algorithm design, mostly in the '70s, in realizing how important this is, that the use of randomization can make algorithms more elegant. Simpler, easier to code, faster, or just simply you can solve problems that you could not solve as easily without the use of randomization. It's really one thing that should be in your toolbox as an algorithm designer, randomization. Quick Sort will be the first [INAUDIBLE] application, but we'll see a couple more later in the course. Now by the end of this sequence of video's I'll have given you a complete rigorous argument about why this works. Why with random pivots, quick sort always runs very quickly, on average. But, you know, before moving into anything to formal let's develop a little bit of intuition or at least kind of a daydream. About why on Earth could this possibly work, why on Earth could this possibly be a good idea, to have randomness internal to our pro sort implementation. Well, so first just very high level, what would be sort of the hope, or the dream. The hope would be, random pivot's are not going to be perfect, I mean you're not going to just sort of guess the median, or you only have a one in chance of figuring out which one the median is, but the hope is that most choices of a pivot will be good enough. So that's pretty fuzzy. Let's drill down a little bit and develop this intuition further. Let me describe it in two steps. The first claim is that, you know in our last quiz we said suppose we get lucky and we always pick the median in every single recursive call. And we observed we'd do great. We'd get end log in running time. So now let's observe that actually to get the end log in running time, it's not important that we magically get the median every single recursive call. If we get any kind of reasonable pivot, by which a pivot that gives us some kind of approximately balanced with the problems, again, we're going to be good. So the last quiz really wasn't particular to getting the exact median. Near medians are also fine. To be concrete, suppose we always pick a pivot which guarantees us a split of 25 to 75, or better. That is, both recursive calls should be called on arrays of size, at most, 75% of the one we started with. So precisely if we always get a 25, 75 split or better in every recursive call I claim that the running time of quick sort in that event will be big O of n log n. Just as it was in the last quiz where we're actually assuming something much stronger where we're getting a median. Now this is not so obvious, the fact that 25, 75 splits guarantee analog and running time. For those of you that are feeling keen you might want to try to prove this. You can prove this using a recursion tree argument, but because you don't have balanced sub problems you have to work a little bit harder than you do in the cases covered by the master method. So that's the first part of the intuition, and this is what we mean by a pivot being good enough. If we get a 25, 75 split or better, we're good to go, we get our desired, our target analog. The second part of the intuition is to realize that actually we don't have to get all that lucky to just be getting a 25, 75 split. That's actually a pretty modest goal and even this modest goal is enough to get n log n running time, right? So suppose our array contains the numbers, the integers between 1 and 100, so it is an array of length 100. Think for a second, which of those elements is going to give us a split that's 25, 75 or better? So, if we pick any element between 26 and 75 inclusive, will be totally good, right? If we pick something that's at least 26, that means the left subproblem is going to have at least the elements 1 through 25. That'll have at least 25% of the elements. If we pick something less than 75 then the right sub-problem will have all of the elements 76 through 100 after we partition, so that'll also have at least 25% of the elements. So anything between 26 and 75 gives us a 75-25 split or better. But that's a full half of the elements, so it's as good as just flipping a fair coin hoping to get heads. So with 50% probability, we get a split that's good enough to get this n log n bound. And so again, the high level hope is that often enough, half of the time, we get these good enough splits, 25-75 split or better, so that would seem to suggest n log n running time on average is a legitimate hope. So that's the high level intuition, but if I were you I would certainly not be content with this somewhat hand wavy explanation that I've given you so far. What I've told you is sort of the hope the dream, why there is at least a chance this might work. But the question remains, and I would encourage such skepticism, which is does this really work? And to answer that we're going to have to do some actual mathematical analysis, and that's what I'm going to show you. I'm going to show you a complete rigorous analysis of the quick sort algorithm with random pivots, and we'll show that yes in fact, it does really work. And this highlights what's going to be a recurring them of this course, and a recurring theme in the study and understanding of algorithms. Which is that quite often there's some fundamental problem you're trying to code with a solution, you come up with a novel idea, it might be brilliant, and it might suck. You have no idea. Now, obviously, you code up the idea, run it on some concrete instances and get a feel for whether it seems like a good idea or not. But if you really want to know fundamentally what makes the idea good or what makes the idea bad, really, you need to turn to mathematical analysis to give you a complete explanation. And that's exactly what we're going to do with QuickSort, and then we'll explain in a very deep way why it works so well. Specifically in the next sequence of three videos I'm going to show you an analysis, a proof of the following theorem about QuickSort. So under no assumptions about the data, that is, for every input array of a given length, say n, the average running time of QuickSort implemented with random pivots is big O of n log n. And again in fact it's theta of n log n but we'll just focus on the big O of n log n part. So this is a very, very cool theorem about this randomized QuickSort algorithm. One thing I want to be clear so that you don't under sell this guarantee in your own mind, this is a worst case guarantee with respect to the input. Okay so notice at the beginning of this theorem what do we say? For every input array of length n, all right? So, we have absolutely no assumptions about the data. This is a totally general purpose sorting separating which you can use whatever you want even if you have no idea where the data is coming from. And these guarantees are still going to be true. This of course is something I held forth about at some length back in our guiding principles video, when I argued that if you can get away with it, what you really want is general purpose algorithms. Which make no data assumption, so they can be used over and over again in all kinds of different contexts and that still have great guarantees, QuickSort is one of those. So basically if you have a data set and it fits in the main memory of your machine, again sorting is a four free sub routine in particular QuickSort. The QuickSort implementation is for free. So this just runs so blazingly fast, doesn't matter what the array is, maybe you don't even know why you want to sort it. But go ahead, why not? Maybe it will make your life easier, like it did for example in the closest pair algorithm for those of you who watch those two optional videos. Now the word average does appear in this theorem and as I've been harping on, this average is not over any assumptions on the data. We're certainly not assuming the the input array is random in any sense. The input array can be anything, so where is the average then coming from? The averaging is coming only from randomness which is internal to our algorithm. Randomness that we put in the code in ourselves, that we're responsible for. So remember, randomized algorithms have the interesting property that even if you run it on the same input over and over again, you're going to get different executions. So the running time of a randomized algorithm can vary as you run it on the same input over and over again. The quizzes have taught us that the running time of QuickSort on a given input fluctuates from anywhere between the best case of n log n to the worst case of n squared. So what this theorem is telling us is that for every possible input array, while the running time does indeed fluctuate between an upper bound of n squared and a lower bound of n log n. The best case is dominating. On average it's n log n, on average it's almost as good as the best case. That's what's so amazing about QuickSort. Those n squared that can pop up once in a while, doesn't matter. You're never going to see it, you're always going to see this n log n like behavior in randomized QuickSort. So for some of you I'll see you next in a video on probability review, that's optional. For the rest of you I'll see you in the analysis of this theorem.

# 6-1-Analysis I - A Decomposition Principle

So this is the first video of three in which we'll mathematically analyze the running time of the randomized implementation of quick sort. So in particular we're going to prove that the average running time of quick sort is big O of n log n. Now this is the first randomized algorithm that we've seen in the course and therefore in its analysis will be the first time that we're going to need any kind of probability theory. So let me just explain upfront what I'm going to expect you to know. In the following analysis. Basically, I need you to know the first few ingredients of discrete probability theory. So I need you to know about sample spaces, that is how to model all of the different things that could happen, all of the ways that random choices could resolve themselves. I need you to know about random variables, functions on sample spaces, which take on real values. I need you to know about expectations that is average values of random variables and very simple but very key propriety we're going to need in the analysis of quick sort is linearity of expectation. So if you haven't seen this before or if you're too rusty definitely you should review this stuff before you watch this video. Some places you can go to get that necessary review you can look at the probability review part one video. That's up on the course's website. If you'd prefer to read something, like I said at the beginning of the course, I recommend the free online lecture notes by Eric Lehman and Tom Leighton, Mathematics for Computer Science. That covers everything we'll need to know, plus much, much more. There's also a Wikibook on Discrete Probability, which is a perfectly fine, obviously, free source in which you can learn the necessary material. Okay? So after you've got that sort of fresh in your mind, then you're ready to watch the rest of this video. And in particular, we're ready to prove the following theorems stated in the previous video. So the quick sort algorithm with a randomized implementation, that is we're in every single recursive subcall, you pick a pivot uniformly at random. We stated the following assertion. But for every single input, so for a worst case input array of length n, the average running time of QuickSort with random pivots is O(n log n). And again, to be clear where the randomness is, the randomness is not in the data. We make no assumptions about the data. As per our guiding principles. No matter what the input array is, averaging only over the randomness in our own code, the randomness internal to our algorithm. We get a running time of n log n. We saw in the past that the best case behavior of QuickSort is n log n. Its worst case behavior is n squared. So this theorem is asserting that no matter what the input array is, the typical behavior of QuickSort is far closer to the best case behavior than it is to the worst case behavior. Okay. So that's what we're going to prove in the next few videos. So let's go ahead and get started. So first I'm going to set up the necessary notation and be clear about what exactly is the sample space, what is the random variable that we care about, and so on. So we're going to fix an arbitrary array of length N. That's going to be the input to the quick sort algorithm. [SOUND]. And we'll be working with this fixed but arbitrary input array for the remainder of the analysis. Okay. So just fix a single input in your mind. Now, what's the relevant sample space? Well, recall what a sample space is. It's just all the possible outcomes of the randomness in the world. So it's all the distinct things that could happen. Now here, the randomness is of our own devising. It's just the random pivot sequences, the random pivots chosen by QuickSort. So omega is just the set of all possible random pivots the QuickSort could choose. Now the whole point of this theorem proving that the average running time of quick sort is small boils down to computing the expectation of a single random variable. So here's the random variable we're going to care about. For a given pivot sequence remember that random variables are real value functions. Defined on the sample space. So for a given point in the sample space or pivot sequence sigma, we're going to define C of sigma as the number of comparisons that quick sort makes. Where by comparison, I don't mean something like with an array index in a for-loop. That's not what I mean by comparison. I mean a comparison between two different entries of the input array, by comparing the third entry in the array against the seventh entry in the array, to see whether the third entry or the seventh entry is smaller. Notice that this is indeed a random variable that is given knowledge of the pivot sequence sigma, the choices of all pivots. You can think of quick sort at that point as just a deterministic algorithm with all of the pivot choices pre-determined, and so a deterministic version of QuickSort make some deterministic member of comparisons so for giving pivot sequence sigma, we're just calling C of sigma to be however many comparisons it makes given those choices of pivots. Now with the theorem I stated is not about the number of comparisons of QuickSort but rather about the running time of QuickSort, but really to think about it kind of the only real work that the QuickSort algorithm does, is make comparisons between pairs of elements in the input array. The axis is a little bit of other book keeping but that's all noise that second over stuff. All QuickSort really does is compare between pairs of elements in the input array. And if you want to know what I mean by that a little more formally, dominated by comparisons, I mean that there exists a constant C so that the total number of operations of any type that QuickSort executes is at most a constant factor larger than the number of comparisons. So lets say that by RT, I mean the number of primitive operations of any form, that QuickSort uses. And for every previd sequence, sigma, the total number of operations, is no more than a constant times the total number of comparisons. And if you want a proof of this it's not that interesting so I'm not going to talk about it here. But in the notes posted on the website there is a sketch of why this is true. How you can formally argue that there isn't much work beyond just the comparisons. But I hope most of you find that to be pretty intuitive. So given this, given that the running time that QuickSort boils down just to the number of comparisons. We want to prove the running time is n log n. All we gotta do, quote unquote, all we have to do this proves that the average number of comparisons the QuickSort mix is all nlogn. And that's what we're going to do. That's what the rest of these lecture is all about. So that's what we got to prove. We got to prove the expectation of this random variable C which counts up the number of comparisons QuickSort mix is for arbitrary input array of link n bound by big O of nlogn So the high order bit of this lecture is a decomposition principle. We've identified this random variable, C, the number of comparisons and it's exactly what we care about. It governs the average running time of QuickSort. The problem is, it's quite complicated. It's very hard to understand what this capital C is, it's fluctuating between nlogn and then squared. And it's hard to know how to get a handle on it. So how are we going to go about proving this assertion, that the expectant number of comparisons that QuickSort makes, is on average just O of nlogn. At this point we've actually have a fair amount of experience with divide and conquer algorithms. You've seen a number of examples. And whenever we had to do a running time analysis of such an algorithm we'd write out a recurrence we applied the master method or in the worst case we'd run our recursion tree to figure out the solution at our recurrence so you'd be very right to expect something similar to happen here. But as we probe deeper and we think about QuickSort we quickly realized that the master method just doesn't apply, or at least not in the form that we're used to, the problem is two fold. So first of all the size of the two sub-problems is random, right? As we discuss in the last video, the quality of the pivot is what determines how balanced the split we get into the two sub-problems. It could be as bad as a sub-problem of size 0 and one of size N minus 1. Or it could be as good as a perfectly balanced split into two sub problems of equal sizes but we don't know. It's going to depend on the random choice of the pivot. Moreover the master method at least as we discussed it required solved subproblems to have the same size and unless you're extremely lucky that's not going to happen. In the QuickSort algorithm. It is possible to develop a theory of recurrence relations for randomized algorithms and apply that to QuickSort in particular. But I'm not going to go that route for two reasons. The first one is't really quite messy. It get's pretty technical to talk about solutions to recurrences for randomized algorithms. Or to thing about random recursion trees, both of those get pretty complicated. The second reason is, I really want to introduce you to what I call a decomposition principle. By which you take a random variable that's complicated, but that you care about a lot. You decompose it into simple random variables, which you don't really care about in their own right, though it's easy analyze. And then you stitch those two things together using linearity and expectation. So that's going to be the workhorse for our analysis of the QuickSort algorithm. And it's going to come up again a couple times in the rest of the course, for example, when we study hashing. So to explain how this decomposition principle applies to QuickSort in particular. I'm going to need to introduce to you the building blocks, simple random variables. Which will make up the complicated random variable that we care about, the number of comparisons. Here's some notation. Recall that we fixed in the background an arbitrary array of length n and that's denoted by capital A. And some notation which is simple but also quite important. By z sub i, what I mean is the ith smallest element in the input array capital A, also know as the ith order statistic. So let me tell you what zi is not. What zi is not, in general, is the element in the ith position of the input unsorted array. What zi is, is it's the element which is going to wind up in the ith element of the array, once we sort it. Okay, so if you fast forward to the end of a sorting algorithm and position i, you're going to find zi. So, let me give you an example. So suppose we had just a simple array here, unsorted with the numbers 6, 8, 10 and 2. Then z1, well that's the first smallest, the one smallest, or just the minimum. So z1 would be the 2, z2 would be the 6, z3 would the the 8 and z4 would be the 10, for this particular input array. Okay, so zi is just the ith smallest number. Whatever it may lie on the original unsorted array, that's what zi refers to. So we already defined the sample space. That's just all possible choices of pivots the QuickSort might make. I already described one random variable, the number of comparisons that QuickSort makes on a particular choice of pivots. Now I'm going to introduce a family of much simpler random variables. Which count merely the comparisons involving a given pair of elements in the input array, not all elements, just a given pair. So for a given a choice of pivots, a given sigma, and for given choices of inj, both of which are between 1 and n. And so we only count things once, so I'm going to insist the i is less than j always. And now here's a definition, my xij and this is a random variable, so it's a function of the pivots chosen. This is going to be the number of times that zi and zj are compared in the execution of QuickSort. Okay, so this is going to be an important definition in our analysis. It's important you understand it. So, for something like the third smallest element and the seventh smallest element. xij is asking, that's when i equals 3 and j equals 7, x37 is asking how many times those two elements get compared as QuickSort proceeds. And this is a random variable in the sense that if the pivot choices are all predetermined, if we think of those being chosen in advance. Then there's just some fixed deterministic number of times that zi and zj get compared. So it's important you understand these random variables xij, so the next quiz is going to ask a basic question about the range of values that a given xij can take on. So for this quiz we're considering as usual some fixed input array. And now furthermore fixed to specific elements of the input array. For example, the third smallest element, wherever it may lie, and the seventh smallest element, wherever it may lie. Think about just these pair of two elements. What is the range of values that the corresponding random variable xij can take on? That is what are the different number of times that a given pair of elements might be conceivably get compared in the execution of the QuickSort algorithm? All right, so the correct answer to this quiz is the second option. This is not a trivial quiz. This is a little tricky to see. So the assertion is that a given pair of elements, they might not be compared at all. They might be compared once and they're not going to get compared more than once. So here what I'm going to discuss is why it's not possible for a given pair of elements to be compared twice during the execution of QuickSort. It'll be clear later on, if it's not already clear now, that both 0 and 1 are legitimate possibilities. A pair of elements might never get compared and they might get compared once. And again, we'll go into more detail on that in the next video. But why is it impossible to be compared twice? Well think about two elements, say the third element and the seventh element. And let's recall how the partition subroutine works. Observe that in QuickSort, the only place in the code where comparisons between pairs of input array elements happens. It only happens in the partition subroutine, so that's where we have to drill down. So what are the comparisons that get made in the partition subroutine? Well, go back and look at that code. The pivot element is compared to each other element in the input array exactly once. So the pivot just hangs up in the first entry of the array. We have this for loop, this index j which marches over the rest of the array. And for each value of j, the jth element of the input array gets compared to the pivot. So summarizing, in an invocation of partition, every single comparison involves the pivot element. So two elements get compared if and only if one is the pivot. All right so let's go back to the question. Why can't a given pair of elements of the input array get compared two or more times? Well, think about the first time they ever get compared in QuickSort. It must be the case, that at that moment we're in a recursive call where either one of those two is the pivot element. So if it's the third smallest element or the seventh smallest element. The first time those two elements are compared to each other, either the third smallest or the seventh smallest is currently the pivot. Because all comparisons involve a pivot element. Therefore, what's going to happen in the recursion, well the pivot is excluded from both recursive calls. So, for example, if the seventh smallest element is currently the pivot, that's not going to be passed on the recursive call which contains the third smallest element. Therefore if you're compared once, one of the elements is the pivot and they'll never be compared again, because the pivot will not even show up in any future recursive calls. So let me just remind you of some terminology. So a random variable which can only take on the values 0 or 1 is often called an indicator random variable, because it's just indicating whether or not a certain things happens. So, in that terminology, each xij is indicating whether or not the ith smallest element in the array and the jth smallest element in the array ever get compared. It can't happen more than once, it may or may not happen, and xij is 1 precisely when it happens. So that's the event that it's indicating. Having defined the building blocks I need, these indicator random variables, these xij's. Now I can introduce you to the decomposition principle as applied to QuickSort. So there's a random variable that we really care about, which is denoted capital C, the number of comparisons the QuickSort makes. That's really hard to get a handle on, in and of itself, but we can express C as the sum of indicator random variables, of these xijs. And those we don't care about in their own right, but they're going to be much easier to understand. So let me just rewrite the definitions of C in the xij, so we're all clear on them. So c, recall, counts all of the comparisons between pairs of input elements that QuickSort makes, whereas an xij only counts the number. And it's going to be 0 or 1, comparisons that involve the ith smallest and the jth smallest elements in particular. Now, since every comparison involves precisely one pair of elements, some i and some j with i less than j, we can write c as the sum of the xijs. So don't get intimidated by this fancy double sum. All this is doing is it's iterating over all of the ordered pairs, so all of the pairs ij, where i and j are both between 1 and n and where i is strictly less than n. This double sum is just a convenient way to do that iteration. And of course, no matter what the pivots chosen are, we have this equality, okay? The comparisons are somehow split up amongst the various pairs of elements, the various is and js. Why is it useful to express a complicated random variable as a sum of simple random variables? Well, because an equation like this is now right in the wheelhouse of linearity of expectation, so let's just go ahead and apply that. Remember, and this is super, super important, linearity of expectation says that the expectation of a sum equals the sum of the expectations. And moreover, this is true whether or not the random variables are independent, okay? And I'm not going to prove it here, but you might want to think about the fact that the xijs are not, in fact, independent. So we're using the fact that linear expectation works even for non-independent random variables. Again, why is this interesting? Well, the left hand side, This is complicated, right? This is some crazy number of comparisons by some algorithm on some arbitrarily long array. And it fluctuates between two pretty far apart numbers n log n and n squared. On the other hand, this does not seem as intimidating. Given xij, it's just 0 or 1, whether or not these two guys get compared or not. So that is the power of this decomposition approach, okay? So, it reduces understanding a complicated random variable to understanding simple random variables. In fact, because these are indicator random variables, we can even clean up this expression some more. So for any given xij being a 0, 1 random variable, if we expand the definition of expectation, just as an average over the various values, what is it? Well, it's some probability it takes on the value 0, that's possible, and then some possibility it takes on the value 1. And of course, this 0 part, we can very satisfyingly delete, cancel. And so, the expected value of a given xij is just the probability that xij = 1. And remember, it's an indicator random variable. It's 1 precisely when the ith smallest and the jth smallest elements get compared. So putting it all together, we find that what we care about. The average value of the number of comparisons made by QuickSort on this input array is this double sum, which literates over all ordered pairs, where each sum and is the probability that the corresponding xij = 1. That is the probability that zi and zj get compared. And this is essentially the stopping point for this video for the first part of the analysis, so let's call this star and put a nice circle around it. So what's going to happen next is that in the second video for the analysis, we're going to drill down on this probability, probability that a given pair of elements gets compared, and we're going to nail it. We're going to give an exact expression as a function of i and j for exactly what this probability is. Then in the third video, we're going to take that exact expression, plug it into the sum, and then evaluate this sum. And it turns out the sum will evaluate to O of n log n. So that's the plan. That's how you'll apply decomposition in terms of 0, 1 or indicator random variables, apply linearity of expectation. In the next video, we'll understand these simple random variables, and then we'll wrap up in the third video. Before we move on to the next part of the analysis, I do just want to emphasize that this decomposition principle is relevant not only for QuickSort, but it's relevant for the analysis of lots of randomized algorithms. And we will see more applications, at least one more application, later in the course. So just to kind of really hammer the point home, let me spell out the key steps for the general decomposition principle. So first you need to figure out what is it you care about. So in QuickSort, we cared about the number of comparisons. We had this lemma that said the running time is dominated by comparisons. So we understood what we wanted to know, the average value for the number of comparisons. The second step is to express this random variable y as a sum of simple random variables, ideally indicator or 0, 1 random variables. Now you're in the wheel house of linearity of expectation, you just apply it, and you find that what it is you care about, the average value of the random variable y is just the sum of the probabilities of various events. That given xl, random variable is equal to 1. And so the upshot is to understand the seemingly very complicated left-hand side, all you have to do is understand something, which in many cases, is much simpler, which is understand the probability of these various events. In the next video, I'll show you exactly how that's done in the case of QuickSort, where we care about the xijs, the probability that two elements gets compared. So let's move on and get exact expression for that probability.

# 6-2-Analysis II - The Key Insight

This is the second video of three, in which we prove that the average running time of randomized quicksort is big O of n log n. So, to remind you of the formal statements. So again we're thinking about quicksort where we implement the choose pivot sub routine to always choose a pivot uniformly at random from the sub array that it gets passed. And we're proving that for a worst case input array for an arbitrary input array of length n, the average running time of quicksort where the average is over the random pivot choices is big O of n log n. So let me remind you of the story so far. This is where we left things at the previous video. We defined a few random variables. The sample space, recall, is just the, all of the different things that could happen, that is all of the random coin flip outcomes that quicksort could produce. Which is equivalent to all of the pivot choices made by quicksort. Now, the random variables we care about. So first of all, there is C. Which is the number of comparisons between pairs of elements in the input array that quicksort makes for a given pivot sequence sigma. And then there are the xij's. And so that's just meant to count the number of comparisons involving the ith smallest and the jth smallest elements in the input array. Where you recall that zi and zj denote the ith smallest and jth smallest entries in the array. Now because every comparison involves some zi and some zj we can express C as a sum over the xij's. So we did that in the last video, we applied linearity at expectation, we used the fact that xij are zero one, that is indicator random variables to denote that to write the expectation of an xij just as the probability that it's equal to one and that gave us the following expression. So the key insight and really the heart of the quicksort analysis is to derive an exact expression for this probability as a function of i and j. So for example if the third smallest element in the array, the seventh smallest element in the array. Wherever they may be scattered in the input array we want to know exactly what's the probability that they get compared at some point in the execution of quicksort. And we're going to get a extremely precise understanding of this probability in the form of this key claim. So for all pairs of elements, and again, ordered pairs. So we're thinking of i being less than j. The probability that zi and zj get compared at some point in the execution of quicksort is exactly 2 divided by j- i + 1. So for example in this example of the third smallest element and the seventh smallest element, it would be exactly 40% of the time, two over five is how often those two elements would get compared if you ran quicksort with a random choice of pivots, and that's going to be true for every j and i. The proof of this key claim is the purpose of this video. So how do we prove this key claim? How do we prove that the probability that zi, zj get compared is exactly 2 over quantity j- i +s 1. Well fix your favorite ordered pair, so fix elements zi, zj with i less than j, for example the third smallest and the seventh smallest element in the array. Now, what we want to reason about is the set of all elements in the input array between zi and zj inclusive. And I don't mean between in terms of positions in the array, I mean between in terms of their values. So consider the set between zi and zj + 1 inclusive, so zi, zi + 1,... Zj- 1, Zj. So for example, the third, fourth, fifth, sixth and seventh smallest elements in the input array. Wherever they may be, okay. And of course, the initial array is not sorted, so there's no reason to believe that these j minus i plus 1 elements are contiguous, okay. They're scattered throughout the input array. But we're going to think about them, okay, zi through zj inclusive. Now throughout the execution of quicksort, these j minus i plus 1 elements lead parallel lives at least for awhile in the following sense. Begin with the outermost call to quicksort and suppose that none of these j minus i plus 1 elements is chosen as a pivot. Where then could the pivot lie? Well it can only be a pivot that's greater than all of these or it could be less than all of these. For example if this is the third fourth, fifth, sixth and seventh smallest elements in the array, well the pivot is either the minimum or the second minimum in which case it's smaller than all five elements, or it's the eighth or largest, or larger elements in the array in which case it's bigger than all of them. There's no way you're going to have a pivot that somehow is wedged in between this set because this is a contiguous set of order statistics, okay. Now what do I mean by these elements leading parallel lives? Well, in the case where the pivot is chosen to be smaller than all of these elements, then all of these elements will wind up to the right of the pivot. And they will all be passed to a common recursive call. The second recursive call. If the pivot is chosen to be bigger than all of these elements, then they'll all show up on the left side of the partitioned array. And they'll all be passed to the first recursive call. Iterating this or proceeding inductively, We see that as long as the pivot is not drawn from the set of j minus i plus 1 elements. This entire set will get passed on to the same recursive call. So these j minus i plus 1 elements are living blissfully together in harmony until the point in which one of them gets chosen as a pivot. And that, of course, has to happen at some point. The recursion only stops when the array length is equal to zero or one. So, if for no other reason at some point there will be no other elements in a recursive call other than these j minus i plus 1, okay. So at some point, the reverie is interrupted and one of them is chosen as a pivot. So let's pause the quicksort algorithm and think about what things look like at the time when one of these j minus i plus 1 elements is first chosen as a pivot element. There are two cases worth distinguishing between. In the first case the pivot happens to be either zi or zj. Now remember what it is we're trying to analyze. We're trying to analyze the frequency, the probability with when zi and zj gets compared. Well, if zi and zj are in the same recursive call, and one of them gets chosen as the pivot, then they're definitely going to get compared. Remember when you partition an array around this pivot element, the pivot get's compare to everything else. So if zi's chosen as a pivot, it certainly get's compare to zj. If zj gets chosen as a pivot, it gets compared to zi. So either way if one of these two is chosen, they're definitely compared. If, on the other hand, the first of these j minus i plus 1 elements to be chosen as a pivot is not zi or zj. If, instead, it comes from the set zi plus 1, so on, up to zj minus 1. Then the opposite is true. Then zi and zj are not compared now. Nor will they ever be compared in the future. So why is that? Well that requires two observations. First recall that when you choose a pivot and you partition an array, all of the comparisons involve the pivot. So two elements neither of which is the pivot do not get compared in a partition sub routine. So they don't get compared right now. Moreover, since zi is the smallest of these and Zj is the biggest of these, and the pivot comes from somewhere between them. This choice of pivot will split zi and zj into different recursive calls, zi gets passed to the first recursive call, zj gets passed to the second recursive call and they will never meet again. So there's no comparison's in the future, either. So these two observations right here I would say is the key insight in the quicksort analysis. The fact that for a given pair of elements we can very simply characterize exactly when they get compared and when they do not get compared in the quicksort algorithm. That is they get compared exactly when one of them is chosen as the pivot before any of the other elements with value in between those two has had the opportunity to be a pivot. That's exactly when they get compared. So this will allow us to prove this key claim, this exact expression on the comparison probability. That will plug into the formula we had earlier and will give us the desired bound on the average number of comparisons. So let's fill in those details. So first let me rewrite the high order bit from the previous slide. So now at last we will use the fact that our quicksort implementation always chooses a pivot uniformly at random. That each element of a sub array is equally likely to serve as the pivot element in the corresponding partition call. So what does this buy us? This just says all of the elements are symmetric. So each of the elements zi, zi plus 1, all the way to zj, is equally likely to be the first one asked to serve as a pivot element. Now the probability that zi and zj get compared is simply the probability that we're in case one, as opposed to in case two. And since each element is equally likely to be the pivot, that just means there's sort of two bad cases, two cases in which one can occur out of the j minus i plus 1 possible different choices of pivot. Now we're talking about a set of j minus i plus 1 elements. Each of whom is equally likely to be asked to be served first as a pivot element. And the bad case, the case that leads to a comparison, there's two different possibilities for that. It was zi or zj is first. And the other j minus i minus 1 outcomes lead to the good case where zi and zj never get compared. So overall, because everybody's equally likely to be the first pivot, we have that the probability with zi and zj get compared. Is exactly the number of pivot choices that lead to comparison, divided by the number of pivot choices overall. And that is exactly the key claim. That is exactly what we asserted was the probability that a given zi and zj get compared for no matter what i and j are. So, wrapping up this video, where does that leave us? We can now plug in this expression for this probability of comparison probabilities. Into the double sum that we had before. So putting it all together what we have is that what we really care about the average number of comparisons that quicksort makes on this particular input of array n, of length n is just this double sum which iterates over all possible ordered pairs ij. And what we had here before was the probability of comparing zi and zj we now know exactly what that is so we just substitute. And this is where we're going to stop for this video so this is going to be our key expression star which we still need to evaluate but that's going to be the third video. So essentially we've done all of the conceptual difficulty in understanding where comparisons come from in the quicksort algorithm. All that remains is a little bit of an algebraic manipulation to show that this starred expression really is big O log n. And that's coming up next.

# 6-3-Analysis III Final Calculations

So we're almost at the finish line of our analysis of quick sort. Let me remind you what we're proving. We're proving that for the randomized implementation of quick sort where we always choose the pivot element to partition around uniformly at random, we're showing that for every array, every input of length N, the average running time of quick sort over the random choices of pivots is [inaudible] of N log N. So we've done a lot of work in the last couple of videos. Let me just remind you about the stories so far. In the first video what we did is we identified the relevant random variable that we cared about, capital C, the number of comparisons that Quicksort makes among the pairs of elements in the input array. Then we applied the decomposition approach. We expressed capital C, the overall number of comparisons, as a sum of indicator or 0-1 random variables. For each of those variables XIJ, just counted the number of comparisons involving the Ith smallest and Jth smallest entries in the array, and that's gonna be either zero or one. Then we applied linearity of expectation to realize, all we really needed to understand was the comparison probabilities for different pairs of elements. [inaudible]. Second video we nailed what that comparison probability is, specifically, for the I smallest and the J smallest elements in the array, the probability that quick sort compares them when you always make random [inaudible] choices is exactly. Two divided by the quantity J minus I. Plus one. So putting that all together, yields the following expression, governing the average number of comparisons made by quick sort. One thing I want you to appreciate is, is in the last couple of videos, we've been sort of amazingly exact as algorithmic analysis goes. Specifically we've done nothing sloppy whatsoever. We've done no estimates. The number of comparisons that quick store makes on average is exactly this double sum. Now surely we'll do some inequalities to make our lives a little bit easier. But up to this point everything has been completely exact. And this will actually see why there's small constants in the, in the, in quick sort. It's basically going to be this factor two. Now the next question to ask is, what are we shooting for? Remember the theorem we want to prove is that the expected number of comparisons really the expected run time is all of N log N, so we're already done. Well not quite we're gonna have to be a little bit clever, so if we're looking at this double sum, and we ask how big are the sum ends and how many terms are there? Well the biggest sum ends we're ever going to see are when I and J are right next to each other when J is one bigger than I, and in that case this fraction is gonna be one half. So the terms can be as big as one half, how many terms are there? Well there's a quadratic number of terms. So it would be very easy to derive an upper bound that's quadratic in N, but that's not what we want. We want one that's N log N. So to drive that, we're gonna have to be a little bit more clever about how we evaluate this sum. So, the idea is, what we're going to do, is to think about a fixed value of I in this outermost sum. And then we're gonna ask, how big could the inner sum be? So let's fix some value of I, the value of the index in the outer sum. And then let's look at the inner sum, where J ranges from I plus one up to N, and the value of the sum end is one over the quantity J minus I plus one. So how big can this be? Well, let's first understand what the terms actually are. So J starts at I plus one and then it ascends to N. And as J gets bigger the denominator gets bigger. So the sum ends get smaller. So the biggest sum end is gonna be the very first one. And J is as small as possible. Namely I plus one. When J is I plus one the sum end is one half. Then J gets incremented in the sum. And so that's, we're gonna pick up a one third term followed by one fourth term, and so on. So there's gonna be, for every inner sum is gonna have a this form, one-half plus one-half equals one-fourth. And then it's gonna sort of run out at some point, when J equals N. And the biggest term we're ever going to see is gonna be a one over N, in the case where I equals one. So. Let's make our lives easier by taking this expression we started with. Star, and instead of having a double sum, let's just upper bound this with a single sum. So what are the ingredients of a single sum? Well, there's this two, can't forget the two. Then there's N choices for I, actually, there's N minus one choices for I, but let's just be sloppy and say N choices. So that gives us a factor N. And then how big can an inner sum be? Well, inner sum is just a bunch of these terms, one-half plus one-third and so on. The biggest of those inner sums is the one occurring when I equals one, at W, at which point the last term is one over N. So, we're gonna just do a change of variable and express the inner [inaudible], upper bound on each inner sum as the sum from K equal two to N of one over K. So that's looking more manageable just having the single sum involving this index K, and life's gonna get really good when we prove the next claim, which is that this sum cannot be very big, it's only logarithmic in N, even though there's a linear number of sum N's, the overall value of the sum is only logarithmic. That, of course, is gonna complete the proof, 'cause that'll give us an overall bound of two times N times the natural log on N. So it's an N login bound with really quite reasonable constants. So, why is this true? Why is this sum only logarithmically large? Well, let's do a proof by a picture. I'm going to write this sum. In a geometric fashion. So on the X axis, let me mark off points corresponding to the positive integers. And on the Y axis, let me mark off points corresponding to fractions of the form, one over K. And what I?m gonna do is gonna draw a bunch of rectangles. Of decreasing area, specifically they all have with one, and the heights are gonna be like one over K. So the area of this guy's one, the area of this guy's one half, the area of this guy's one third, and so on. And now I'm going to overlay on this picture the graph of the function, the continuous function, F of X equals one over X. So notice that is going to go through these three points. It's gonna kiss all of these rectangles on their upper right corners. Now what is it we're trying to prove? The claim we're trying to prove is that this sum, one half plus one third and so on, is upper bounded by something, so the sum can be just thought of as the areas in these rectangles, the one half, the one third and so on, and we're going to upper bound it by the area under the blue curve, if you notice the area under the blue curve is at least as big as the sum of the areas of the rectangles because the curve hits each of these rectangles in its north east corner. So putting that into mathematics, the sum from K equal two to N of one over K. Is met in above by the integral. And we'll start the area of the curve at one. And then we need it to go all the way up to N. Of the function one over X. The X, so that's the area under the curve. And if you remember a little bit of calculus the integral of one over X is the natural log of X. So this equals the natural log of X. Evaluated at one. Also known as login minus log one. And of course log one would be zero, so that gives us our login. So that completes the proof of the claim. That indeed, the sum of these one over K's is bounded above by the natural log of N, and that in fact completes the proof of the theorem. You've got to be the expected number of comparisons, at most two N times this sum, which is at most log N. And altogether, we find that the expected number of comparisons that quick sort makes on an arbitrary input of length N. Is two times N times the natural log of N. So that would be big o of N, log N, with quite reasonable constants. Now, this is just the number of comparisons, but as we observed earlier, the running time of Quicksort on average is not much more than that, the running time is dominated by the number of comparisons that it makes. Moreover, as we discussed when we were talking about the details of the implementation, it works in place, essentially no extra storage is necessary. So that is a complete and mathematically rigorous explanation of just why Quicksort. Is so quick.

# 7-1-Probability Review I

Welcome to part one of our probability review. The first time that we need these concepts in the course, is for those of you who want to understand the analysis of Quicksort. Why it runs in big O of n log n time on average. And these topics will also come up a couple of other times in the course. For example when we study a randomized algorithm for the minimum cut program in graphs and also when we try to understand the performance of hashing. Here are the topics we're going to cover. We'll start at the beginning with sample spaces and then we'll discuss events and their probabilities. We'll talk about random variables, which are real valued functions on a sample space. We'll talk about expectation, which is basically the average value of a random variable. We'll identify and prove a very important property, called the linearity of expectation, which will come up over and over again. In our analyses of randomized processes. So that's going to be the topics for part one. Then we'll conclude the video with one example tying these concepts together in load balancing. And this video is by no means the only source you can turn to to learn about these concepts. A couple of other sources I recommend are the online lecture notes by Eric Lehman and Tom Leighton. Also, there's a Wikibook on discrete probability, which you could check out. And I want to be clear this is really not meant to be a course or a tutorial on probability concepts, it's really only meant to be a refresher. So I'm going to go at a reasonably fast pace and it's going to be a pretty cursory presentation. And if you want a more thorough review, I would check out one of these other sources. Or your favorite book on Discrete Probability. And along those same lines, I'm thinking that many of you have seen some of this material before. Don't feel compelled to watch this video straight from the beginning to the end. Feel free to just sort of dip in and review the concepts that you need a refresher on. So, let's start at the beginning with sample spaces. So what is a sample space? Well, we're analyzing random processes so any number of things could happen. And in the sample space is just the collection of all of the things that could happen. So this is basically the universe in which we're going to discuss probabilities and average values. So I'll often use the notation big omega to describe the sample space. So one thing we've got going for us in the design of algorithms is typically we can take omega to be a finite set. So that's why we're dealing only with discrete probability which is a very happy thing. because that's much more elementary than more general probability. In addition to defining the outcomes, everything that could possibly happen, we need to define what is the probability of each individual outcome. So of course the probability of each outcome should be at least zero, should be non-negative. And there's also the obvious constraint that the sum of the probabilities should be one. So exactly one thing is going to happen. Now I realize this is a super abstract concept and the next few definitions are also a little abstract. So throughout them I'm going to use two really simple, really concrete examples to illustrate what these concepts mean. So the first example is just going to be you take two six sided dice and you roll them. And of course, the same space is just the 36 different outcomes you could have of these two dice. And assuming that each of these two dice is well crafted, then we expect each of these 36 outcomes to be equally likely, to occur with a probability of one over 36. The second running example I'm going to use is more directly related to algorithms, and it's motivated by the quick sort algorithm. Recall that we're studying the implementation of Quicksort that chooses a pivot, uniformly a random in every recursive call. So, let's just focus on the very first outer most call of Quicksort and think about the random choice of the pivot just in that call. So, then in the sample space all of the different things that could happen is just all of the end different choices for a pivot assuming the array has length n. So we can represent the sample space just as the integer is one two all the way up to N corresponding to the array index of the randomly chosen pivot. And again by definition by the def construction of our code each of these things is equal to likely probability of one over N. Now let's talk about events. An events is nothing more than a subsets of all of the things that could happen, that is a subset of the sample space omega. The probability of an event isn't exactly what you think it would be, it's just the sum of the probabilities of all of the outcomes contained in that event. Right, so an event is just a bunch of stuff that might happen. We know the probability of each individual thing that can happen, we add them up to get the probability of an event. So the next two quizzes are meant to give you some practice with these concepts. And in particular, they'll ask you to compute the probability of events in our two running examples. So on the first quiz, this is our first running example where we think about two dice and we have our 36 possible outcomes. Consider the subset of outcomes in which the sum of the two dice equals 7. What is the probability of that event? Right so the correct answer is the third one. The probability is 1/ 6. Why is that? Well first let's be more precise about what this event is. What are the outcomes in which the sum of the dice is equal to 7? Well there's exactly six such outcomes. 1,6 2,5 3,4 4,3 5,2 and 6,1. Each of the 36 outcomes is equally likely, has the probability of one over 36. So we have six members of the set. Each has probability of one over 36. So the probability is 1/6. Let's move onto the next quiz, which considers our second running example, namely, the randomly chosen pivot. And the outermost call to QuickSort on an input array of length n. So recall that in quick sort, when you choose a pivot, you then partition the array around the pivot. And this splits the input array into two sub-arrays. A left one. Elements less than the pivot. And a right one, those bigger than the pivot. And the more balanced the split into theses two sub problems the better. So ideally we'd like a 50 50 split. So what this quiz asked you is what fraction of pivots, that is what's the probability that a randomly chosen pivot will give you a reasonably good split? Meaning both of the sub problems have size at least 25%. That is you get a split 25, 75 or better. That's what this quiz asks about. What's the probability that your randomly chosen pivot satisfies that property? So the correct answer to this quiz is again the third option. It's a 50% probability you get a 25-75 split or better. So to see why let's again be precise about what is the event that we're talking about. Then we'll compute its probability. So when does a pivot give you a 25-75 split or better? Well for concreteness, suppose the array contained just the integers between one and 100. Now, what's the property we want? We want that both of the two subarrays have at least 25% of the elements, neither one has more than 75% of the elements. Well, if we choose an element that's 26 or bigger in value. Then the left sub-problem will have at least 25 elements, the numbers 1 through 25. And if we choose an element that's at most 75, then the right subarray is going to have at least 25 elements, namely the numbers 76 to 100. So anything between 26 and 75, inclusive, is going to give us a 25-75 split. More generally, any pivot from the middle 50% of the quantiles, is going to give us the desired split. So we do badly if we get something within the first quarter, we do badly if we get something within the last quarter. Anything in the middle works. So more formally, we can say that the event s that we're analyzing is among the possible pivot choices. We're interested in the ones that is not in the first quarter and not in the last quarter. Now the cardinality of this the number of pivots in this set is essentially half of the overall number of pivot choices. I'm ignoring fractions here for simplicity. The probability of this event is the cardinality of this times the probability of each of the individual outcomes. And since we choose the pivot uniformly at random, each one has a probability of one over n. So you get n/2 / n, or 1/2. Now that we've explored the concept of events in our one or two examples. We see that the probability that the sum of two dice is equal to 1/6. A useful fact to know if you're ever playing craps. We know that a pivot gives us a 25-75 split or better in a randomized quick sort with 50% probability. A useful fact if you want to develop intuition for why quick sort is, in fact, quick. That's events. Let's move on to random variables. Random variables are basically some statistic measuring what happens in the random outcome. Formally, if we want to define it. It's a real-valued function defined on the sample space omega. Given an outcome, given a realization of the randomness. This gives you back a number. The random variable that we most often care about in algorithm design is the running time of a randomized algorithm. That's the case, for example, with the quick sort algorithm. Notice, that is, in fact, a random variable. If we know the state of the world. If we know the outcome of all the coin flips that our code's going to make. Then there's just some running time of our algorithm. So, in that sense, it's a random variable. Given the outcomes of the coin flips, out pops a number. The running time, say, in milliseconds, of the algorithm. Here, I'm going to give you a couple more modest examples of random variables in our two running examples. If we're rolling two dice. One very simple random variable takes as input the outcome, the result of the two dice. And spits out the sum. That's certainly a random variable. On any given outcome, it's going to take on some some integer value between 2, at the minimum, and 12, at the maximum. Our second running example is the randomly chosen pivot made by the outermost call to quick sort. Let's think about the random variable, which is the size. Meaning the subarray length, passed to the first recursive call. Equivalently, this random variable is the number of elements of the input array smaller than the randomly chosen pivot. This is a random variable that takes on some interval value between zero, at the smallest. That's if we happen to pick the pivot equal to the minimum of the array. And n-1 at the largest. That's if we happen to pick the maximum element as the pivot element. Next, let's talk about the expectation of a random variable. This is really nothing more than the average. Of course, when you take the average of some statistic. You want to do it weighted by the probability of its various values. Let's just make that precise real quick. Consider some random variable, X. The expectation, this is also called the expected value. And the notation is capital E, square bracket, then of the random variable. Again, in English, the expectation is just the average value. Naturally weighted by the probability of the various possible outcomes. Or more mathematically, we sum over everything that could happen. So let i denote one possible outcome. We look at the value of this random variable when that outcome occurs. And then we weight up times the probability of that outcome occurring. The next two quizzes ask you to compute the expectation of the two random variables that we identified on the previous slide. The first quiz is about two dice. And the random variable, which is the sum of the values of those two dice. What is the average of that random variable? What is its expectation? The answer to this question is the second option. The average value is 7. There's a bunch of different ways to see that. In my opinion, the best way to compute this is using linearity of expectation. Which is the next concept we're going to cover. If you wanted to, you could just compute this by brute force. By which I mean, you could iterate over all 36 possible outcomes. Look at the value of the two dice in each. And just evaluate that sum we had in the definition on the last slide. A slightly sneakier way to do it, if you don't know linearity of expectation. Would be to pair up the various outcomes. So it's equally likely that the sum of the two dice is 2 or 12. It's equally likely to be 3 or 11, 4 and 10, and so on. Each way of pairing up these values of the two dice results in 14. When you average, you get 7. But, again, the right way to do this is linearity of expectation. Which we'll cover next. The second quiz covers the second random variable we identified. Now we're back to QuickSort. And the random pivot chosen in the outermost call. The question is, how big, on average, an expectation is the subarray in the first recursive call? Equivalently, on average, how many elements are going to be less than the randomly chosen pivot? The correct answer to this quiz is the third option. In fact, it's actually quantity n-1 / 2, not n/2. But, basically, half the elements. Again, this sort of a sneaky way to see this if you want. Which is that, clearly, the two recursive calls are symmetric. The expected value of the left recursive call is going to be the same as the expected size of the right recursive call. The two recursive calls always comprise n-1 of the elements. Because they're symmetric, you expect half in each. So n-1 / 2 in each. Though for this problem, I think it's perfectly fine just to compute this using the definition of expectation. If we let X denote the random variable that we care about, the subarray size. Then we can just compute directly by summing over all of the possible outcomes. All of the possible choices of the pivot. With probability 1/n, we choose the minimum of the pivot. Resulting in 0 elements being passed to the first recursive call. With probability 1/n, we pick the second smallest element. Resulting in 1 element being passed to the first recursive call. With probability 1/n, we pick the third smallest. Giving us a subarray size of 2. And so on. With probability 1/n, we pick the maximum element. Giving us a subarray size of n-1. If you just compute this sum out, you will get, as expected, n-1 / 2. Expectation is the last definition that I'm going to give you in this part one of the probability review. Next, is our fifth and final concept for this video. Which is Linearity of Expectation. That's not a definition. That's more of a theorem. What is linearity of expectation? This is a very simple property of random variables that's super-super-important. This comes up all the time when we analyze randomized algorithms. And random processes, more generally. What is linearity of expectation? It's the following, very simple claim. Which I'll sometimes denote just by LIN EXP, for short. Suppose you got a bunch of random variables defined on the same sample space. Then, if you want to think of the expected value of the sum of these random variables. It doesn't matter if you take the sum first and then take the expectation. Or if you take expectations first and then sum. That is, the expected value of a sum of random variables is equal to the sum of the expectations of the individual random variables. One of the reasons linearity of expectations is so ubiquitously useful is because it always works. No matter what these random variables are. In particular, even when the random variables are not independent. Now, I haven't defined independent random variables, yet. That will come in part two, the probability review. But hopefully, you have an intuitive sense of what independence means. Things are independent if knowing something about one of the random variables Doesn't influence what you expect from the other random variable. Now I realize the first time you see linearity of expectation it's a little hard to appreciate. So first of all as far as the applications we'll see plenty throughout this course, pretty much every single application of probability that we'll see the analysis will involve linearity of expectation. But it may be hard to appreciate why this is not a tautology. Just symbolically, it may look like it has to be true. But to point out that there is content here, if I replace the sums by products, then this equation would in general be false, if the random variables are not independent. So the same thing is not true about products, it's really about sums. So let me just give you a trivial illustration of linearity of expectation, point out how it really easily allows us to evaluate the sum of two dice. So in our first running example let's introduce the random variables x1 and x2 for the results of the first and second die respectively. Now computing the expected value of a single die is easy. There's only six outcomes to a enumerate over contrast that with the 36 outcomes to enumerate over when we evaluated the sum of the two dies. So the average value of a single die you won't be surprised to hear is 3.5 right? So it ranges integers between 1 and 6 uniformly so 3.5 on average. And now using linearity of expectation, the sum of two dice is simply double the average value of a single one. So in the next slide I'm going to prove this property, prove linearity of expectation, but frankly the proof is pretty trivial, so if you don't care about the proof that's fine you can skip it without loss I'm inclusing just for completeness. And I got to say I don't know of another mathematical statement which is simultaneously so trivial to prove and so unbelievably useful. It's really something remarkable linearity of expectations. So how's the proof go, well honestly we just write out the sum, the definition of an expectation, then we reverse the sums, and we're done. So let me start with the right hand side of the equation. So that was the sum of expectations of the random variables. So now let's just apply the definition of expectation. So it's just a weighted average over the possible outcomes. In that one, instead of summing first over the random variable j, and then over realized outcome i, I'm going to do it in reverse order. I'm going to sum first over the outcome i and then over the random variable j. Now the probability of outcome i is independent of j so we can yank the p(i) outside of that inner sum. But now what have we got? So inside the parentheses we simply have the value of the sum of the xji's, xj's on the outcome i. And then over here, we're just averaging the sum of the xj's with respect to the probabilities, the pi's. So this is just the definition of the expectation of the sum of the random variables. So that's it. So linearity of expectation is really just a reversal of the double sums. Now for those of you that are rusty on these kinds of manipulations I just want to point out, this reversal of the double sum itself is there's nothing complicated at all about what's going on. So if you want a really pedestrian way to think about what's happening, just imagine that we take these sum ends, these xji, pi's. And we just write them out in a grid, where one, or let's just say, the columns are indexed by the random variable j, and the rows are indexed by the outcome i. And in a given cell of this grid, we just write the, sum end, xji times pi. So if you get lost in the notation with these double sums, the point is you can just interpret each of them in terms of this grid. Both of these double sums are nothing more than the sum of the values in all of the cells of this grid. One order of summation just says you group first according to row sums and then sum those up. That's the first summation. The second summation, you first take column sums and then sum those up. But of course it doesn't matter, you just get the result of everything in the grid. Okay, so there's no tricks up my sleeve when I reverse these sums, it's a totally elementary, trivial thing. Okay, so again linearity of expectation, trivial to prove, incredibly useful. Don't forget it. So I want to conclude this video with one final example in order to tie together all of the concepts that we've just learned, or just reviewed. And that's going to be an example about load balancing, assigning processes to servers. But this in fact is quite important for the analysis of hashing that we're going to see toward the end of the course as well. But for now lets just think about the following simple problem. For some integer n, you have n computer processes that have to be assigned to n servers in some way. Now, you're feeling very lazy, okay, so you're just going to take each of these processes and you're just going to assign it to a totally random server, okay with each server equally likely to get a given process. And the question I want to study is does this laziness cost you, at least on average? So if you look at the server, what's the expected load? So let's proceed to the solution, the answer of this question. So before you start talking about expectations one has to be clear about the sample space and what are the probabilities of the various outcomes. So remember the sample space omega just denotes every possible that could happened. So what are we doing for each process we're assigning to a random server, so all of the things that can happen are all of the different assignments of these n processes to these n servers. And if you think about is there are n raised to the n possible outcomes cause you have n choices for each of the n processes. Moreover, because each process is assigned to one of the servers uniformly at random, each of these n to the n assignments is equally likely, probability 1 over n to the n. Now that we have a sample space, we're in a position to define a random variable. And we already know what random variable we care about, we care about the average load of the server. Now all of the servers are exactly the same, so we just have to focus on one server, let's say the first server, and look at the number of processes assigned to it. And if you go back to the problem statement, what we're asked, is to compute the expected value of Y, the expected number of processes assigned to a server. Now of course, in principle, we could go to the definition of expectation and just compute by brute force the sum over all possible outcomes of the value of y and take the average. Unfortunately, there are n to the n different outcomes, and that's a lot. So what could we do other than this brute force computation? Well recall our example of linearity of expectation in the sum of two dice. We observe that instead of computing the sum by enumerating over all 36 outcomes, it was much better to just focus on a single die, compute its expectation and then conclude with linearity of expectation. So we'll do the same thing here. Instead of focusing on the sum y, we'll focus on constituent parts of y. So whether or not a single process gets assigned to the first server. And then we'll get away with that by linearity of expectation. So more precisely, for a given process j let's define xj to be, whether to be 1, if and only if the jth process gets assigned to the first server 0 otherwise. Zero, one random variables like xj are often called indicator random variables. That's because they, in effect, indicate whether or not a certain event occurs. In this case, whether or not the jth process gets assigned to the first server. Why did I make this definition? Well, observe that the total number of processes that gets assigned to the first server is simply the sum, j equal 1 to n of xj, xj says whether or not a given process, the jth process, is on the first server. The total number is just the sum of these over all j. Now, the benefit from this maneuver is we only have to compute the expectation of a extremely simple indicator random variable xj. This is like the win that we got when we were summing up two dice, by instead of having to compute the sum, the expected value of the sum, we just had to focus on the expectation of a single die, that was really easy. Similarly here, the expectation of a single xj is really easy. Specifically, let's write it out just using the definition of the expectation. So the expected value of an xjis well let's group together all the outcomes in which it takes on the value zero. So the contribution of the expectation is zero for all of those outcomes, and then there's the rest of the outcomes where xj takes on the value one and in those cases it contributes one to the expectation. Now obviously we get some happy cancellation happening here with the zero part. And all we have to worry about is the probability that xj takes on the value one. Okay what was xj again, how did we define it? Remember it's the events that, it's 1 exactly when the jth process gets assigned to the first server. How are processes assigned? Well remember the proposed solution assigned to each process to each of the n servers, equally likely with uniform probability. So the probability of the jth process is assigned to the first service is 1 over n. So this leaves us with just the sum from j equal 1 to n of 1 over n. That is we just sum up 1 over n with itself n times, this of course is equal to 1. So at the end of the day what we find is that the expected number of processes assigned to a given server say the first server is just 1. So at least if we only care about averages, we lose very little from this trivial process of randomly spraying the process to the server. On average, any given server has just one process on it. This is characteristic of the role that randomization plays in algorithm design in computer science more generally. Often we can get away with really simple heuristics just by making random choices. Of course, quicksort is one example of that where we get an extremely, prevalently used practically sorting algorithm just by making it randomly chosen pivets in every recursive call.

# 7-2-Probability Review II

So welcome to part two of our probability review. This video assumes you've already watched part one or at least are familiar with concepts covered in part one. Namely sample spaces, events, random variables, expectation and linearity of expectation. In this part of the review we're going to be covering just two topics. Conditional probability and the closer related topic of independence. Both between events and between random variables. I want to remind you that this is by no means the only source you can or should use to learn this material. A couple of other sources free that I recommend are lecture notes that you can find online by Eric. And also there's a wiki book on discrete probability. So, conditional probability, I hope you're not surprised to hear, is fundamental to understanding randomized algorithms. That said, in the five weeks we have here, we'll probably only use it once. And that's in analyzing the correctness of the random contraction algorithm for computing the minimum cut of an undirected graph. So, just to make sure we're all on the same page, here's some stuff you should have learned, from part one of the probability review. You should know what a sample space is. This represents all of the different outcomes of the random coin flips, all of the different things that could happen. Often in randomized algorithm analysis, this is just all of the possible random choices that the algorithm might make. Each outcome has some known probability [inaudible]. By and, of course, the sum of the probabilities equal one and remember that event is nothing more than a subset of omega. Omega is everything that could possibly happen. S is some subset of things that might happen and, of course, the probability of event is just the probability of, of all the outcomes that the event contains. So, let's talk about conditional probability. So one discusses the conditional probability of one event given a second event. So, let X and Y denote two events, subsets of the same sample space. You might want to think about these two events X and Y in terms of an event diagram. So we could draw a box, representing everything that could conceivably happen. So that's Omega. Then we can draw a blob corresponding to the event X. So that's some stuff. Might or might not happen, who knows. And then the other event Y is some other stuff which might or might not happen. And in general these two events could be disjoint, that is they could have no intersection. Or they might have a non-trivial intersection. X intersect Y. Similarly they need not cover omega. It's possible that nothing X nor Y happens. So what's we're looking to define is the probability of the event X given the even Y. So we write probability of X bar Y, phrased X given Y. And the definition is, I think, pretty intuitive. So given Y means we assume that something in Y happened. Originally anything in omega could have happened. We didn't know what. Now we're being told that whatever happened that lies somewhere in Y. So we zoom in on the part of the picture that, in which contains Y. So that's gonna be our denominator. So, our new world is the stuff in Y. That's what we know happened. And now we're interested in the proportion of Y that is filled up with X. So, we're interested in what fraction of Y's area is occupied by stuff in X. So X intersect Y, divided by the probability of Y. That is by definition the conditional probability of X given Y. Let?s turn to a quiz, using our familiar example of rolling two dice. To make sure that the definition of conditional probability makes sense to you. Okay, so the correct answer to this quiz is the third answer. So let's see why that is. So what are the two events that we care about? We want to know the probability of X given Y, where X is the event that at least one die is a one. And Y is the events that the sum of the two dice is seven. Now, the easiest way to explain this is let's zoom in, let's drill down on the Y. Let's figure out exactly which outcomes Y comprises. So the sum of the two dice, being seven, we saw in the first part of the review, there's exactly six outcomes which give rise to the sum seven, namely the ordered pairs one, six. Two five, three four, four three, five two, and six one. Now, remember that the probability. Of x given y is by definition the probability of x intersect y divided by the probability of y. Now, what you notice from this formula is we actually don't care about the probability of x per se or even about the event x per se, just about x intersect y. So, let's just fig, so, now we know why there has to be six outcomes. Which of those also belong to x? Well, x is those where at least one die is one. So, x intersect y is just going to be the one, six and the six, one. Now the probability of each of the 36 possible outcomes is equally likely. So each one is one over 36. So since X intersects Y, has only two outcomes. That's gonna give us two over 36 in the numerator. Since Y has six outcomes, that gives us a six over 36 in the denominator. When you cancel everything out, you're left with a one third. So just applying the definition of conditional probability to the correct definition of the two relevant events, we find that indeed a third of the time is when you have a one condition on the sum of the two being seven. Let's move on to the independence of two events. So. Again we consider two events, x and y. By definition, the events are dependent if and only if the following equation holds. The probability that both of them happen. That is the probability of x intersect y is exactly equal to the probability that x happens times the probability that y happens. So that's a simple innocuous looking definition. Let me re phrase it in a way that it's even more intuitive. So I'll you check this, it's just a some trivial algebra. This equation holds, for the events X and Y, if and only if, this is just using the definition of conditional probability we had on the last slide, if and only if the probability of X given Y, Is exactly the same thing as the probability of x. So, intuitively, knowing that y happens, gives you no information about the probability that x happens. That's the sense in which x and y are independent. And, you should also check that this holds if and only if, the probability of y, given x, equals the probability of y. So, symmetrically, knowing that X has occurs gives you no information, no new information about whether or not Y has occurred. The probability of Y is unaffected by conditioning on X. So at this juncture I feel compelled to issue a warning. Which is, you may feel like you have a good grasp of independence. But, in all likelihood, you do not. For example I rarely feel confident that I have a keen grasp on independence. Of course I use it all the time in my own research and my own work, but it's a very subtle concept. Your intuition about independence is very often wrong, even if you do this for a living. I know of no other source that's created so many bugs in proofs by professional mathematicians and professional computer science researchers as misunderstandings of independence and using intuition instead of the formal definition. So, for those of you without much practice with independence, here's my rule of thumb for whether or not you treat random variables as independent. If things are independent by construction, like, for example, you define it in your algorithm, so the two different things are independent. Then you can proceed with the analysis under the assumption that they're independent. If there's any doubt, if it's not obvious the two things are independent, you might want to, as a rule of thumb, assume that they're dependent until further notice. So the slide after next will give you a new example showing you things which are independent and things which are not independent. But before I do that I wanna talk about independence of random variables rather than just independence of events. So you'll recall a random variable is from the first video on probability review. It's just a real value function from the sample space to the real numbers. So once you know what happens you have some number. The random variable evaluates to some real number. Now, what does it mean for two random variables to be independent? It means the events of the two variables taking on any given pair of values are independent events. So informally, knowing the value taken on by one of the random variables tells you nothing about what value is taken on by the other random variable. Recalling the definition of what it means for two events to be independent, this just means that, the probability that A takes on value little a, B takes on value little b. The probability that both of those happen is just the product of the probabilities that each happens individually. So what's useful about independence of events is that probabilities just multiply. What's useful about independence of random variables is that expectations just multiply. So, we're going to get an analog of linear expectation where we can take, we can interchange an expectation in the product freely, but I want to emphasize this, this interchange of the expectation of the product is valid only for independent random variables and not in general, unlike linear expectations. And we'll see a non example. We'll see how this fails on the next slide for non independent random variables. So, I'll just state it for two random variables, but the same thing holds by induction for any number of random variables. If two random variables are independent, then the expectation of their product. Equals the product of their expectations. And again, do not forget that we need a hypothesis. Remember, linearity of expectations did not have a hypothesis for this statement about products. We do have a hypothesis of them being independent. So why is this true? Well, it's just a straight forward derivation where you follow your nose or write it out here for completeness, but, but I really don't think it's that important. So you start with the expectation of the product. This is just the average value of A times B, of course weighted by the probability of, of any particular value. So the way we're gonna group that sum is we're going to sum over all possible combinations of values, A and B, that capital A and capital B might take on, so that's gonna give us a value of A times B. Times the probability of that big A takes on the value of little a and capital B takes on the value of little b. So that's just by definition where this is the value of the random variable, capital A times capital B and this is the probability that it takes on that value with the values A and B. Now because A and B are independent, this probability factors into the product of the two probabilities. This would not be true if they were not independent. It's true because they're independent. So same sum where all possible joint values of all A and B. You still have A times B. But now we have times the probability that A takes on the value of A times the probability that B takes on the value of B. So now we just need to regroup these terms. So let's first sum over A. Let's yank out all the terms that depend on little a. Notice none of those depend on little b. So we can yank it out in front of the sum over little b. So I have an A times the probability that big A takes on the value of little a. And then the stuff that we haven't yanked out is the sum over b, of b times, little b times the probability that capital B takes on the value little b. And what's here inside the quantity? This is just the definition of the expectation of b. And then what remains after we have factored out the expectation of b? Just this other sum which is the definition of the expectation of a. So, indeed four independents random variables, the expected value of the product is equal to the product of the expectations. Let's now wrap up by tying these concepts together in an example, a simple example that nevertheless illustrates how it can be tricky to figure out what's independent and what's not. So here's the set up. We're going to consider three random variables. X1, X2 and X3. X1 and X2 we choose randomly, so they're equally likely to be zero or one. But X3 is completely determined by X1 and X2. So it's gonna be the XOR of X1 and X2. So XOR stands for exclusive or. So what that means is that if both of the operands are zero, or if both of them are one, then the output is zero. And if exactly one of them is one, exactly one of them is zero, then the output is one. So it's like the logical or function, except that both of the inputs are true, then you output false, okay? So that's exclusive or. Now this is a little hand wavy, when we start talking about probabilities, if we want to be honest about it, we should be explicit about the sample space. So what I mean by this, is that X1 and X2 take on all values, they're equally likely. So we could have a zero, zero or a one zero or a zero one or a one, one and in each of these four cases, X3 is determined by the first two, as the X or, so you get a zero here, a one here, a one here and a zero there. And each of these four outcomes is equally likely. So let me now give you an example of two random variables, which are independent, and a non example. I'll give you two random variables which are not independent. So first, I claim that, if you think that X1 and X3, then they're independent random variables. I'll leave this for you to check [sound]. This may or may not seem counter-intuitive to you. Remember X3 is derived in part from X1. Never the less, X1 and X3, are indeed independent. And why is that true? Well, if you innumerate over the four possible outcomes, you'll notice that all four possible two byte strings occur as values for one and three. So here they're both zero, here they're both one, here you have a zero and one, and here you have a one and zero. So you've got all four of the combinations of probability one over four. So it's just as if X1 and X3 were independent fair coin flips. So that's basically why the claim is true. Now. That's a perhaps counterintuitive example of independent random variables. Let me give you a perhaps counterintuitive example of dependent random variables. Needless to say, this example just scratches the surface and you can find much more devious examples of both independent and non-independents if you look in, say, any good book on discrete probability. So now let?s consider the random variable X1 product X3. And X two and the claim is these are not independent. So this'll give you a formal proof for. The way I'm going to prove this could be slightly sneaky. I'm not going to go back to the definition. I'm not gonna contradict the consequence of the definition. So it's proved that they're not independent all I need to do, is show that the product of the expectations is not the same as the expectations to the products. Remember if they were independent, then we would have that equality. [inaudible] Product of expectations will equal the expectation to products. So if that's false than there's no way these random variables are independent. So the expectation of the product of these two random variables is just the expected value of the product of all three. And then on the other side, we look at The product of the expected value of X1 and X3. And the expected value of X2. So let's start with the expected value of X2. That's pretty easy to see. That is zero half the time and that is one half the time. So the expected value of X2 is going to be one-half. How about the expected value of X1 and X3? Well, from the first claim, we know that X1 and X3 are independent random variables. Therefore, the expected value of their product is just the product of their expectations. Equal this expectations equal to the expected value of X1 times the expected value of X2, excuse me, of X3. And again, X1 is equally likely to be zero or one. So its expected value is a half. X3 is equally likely to be zero or one so its expected value is a half. So the product of their expectations is one-fourth. So the right-hand side here is one-eighth; one-half times one-fourth, so that's an eighth. What about the left-hand side, the expected value of X1 times X3 times X2? Well, let's go back to the sample space. What is the value of the product in the first outcome? Zero. What is the value of the product in the second outcome? Zero. Third outcome? Zero. Forth outcome? Zero. The product of all three random variables is always zero with probability one. Therefore, the expected value, of course, is gonna be zero. So indeed, the expected value of the product of X1, X3 and X2 zero does not equal to the product of the corresponding expectations. So this shows that X1, X3 and X2 are not independent.