

wk9

3-wk1



wk9

# 0wk9-Overview

WELCOME: Welcome to Part 3 of the Algorithms Specialization: Greedy Algorithms, Minimum Spanning Trees, and Dynamic Programming! Like the previous two parts, the course will have four weeks of lectures and assignments, followed by a final exam. Here are the highlights of the course's first week.

TWO MOTIVATING APPLICATIONS: We begin with a fairly non-technical discussion of two motivating applications --- distributed shortest-path routing in the Internet, and sequence alignment --- to build excitement for the tools that you'll acquire later in this course (and also in Part 4 of the specialization).

INTRODUCTION TO GREEDY ALGORITHMS: The focus of this week and the next is the greedy algorithm design paradigm. These two non-technical videos discuss the pros and cons of this paradigm and describe a cool application to the optimal management of the contents of a cache.

A SCHEDULING APPLICATION: Scheduling problems come up all the time (e.g., how should a shared resource be allocated?) and greedy algorithms are often useful for them. We'll discuss a specific success story --- minimizing the weighted sum of completion times of a bunch of tasks --- in detail. The correctness proof furnishes a particularly clean example of an "exchange argument."

PRIM'S MST ALGORITHM: The minimum spanning tree (MST) problem, in addition to enjoying several applications, is a uniquely great problem for the study of greedy algorithms. Unusually, several different greedy algorithms always compute an optimal solution. We begin here with the Dijkstra-esque Prim's algorithm. The correctness proof requires understanding the subtly beautiful structure of cuts in graphs, while its blazingly fast implementation relies on a deft application of the heap data structure.

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

HOMEWORK #1: The first problem set consists of 5 problems, about greedy scheduling algorithms and minimum spanning trees. The first programming assignment asks you to implement some of the algorithms that we've covered, run them on large inputs, and enter the answer. For the seasoned programmers out there looking for an additional challenge, try doing the programming assignments in a programming language that you don't already know!

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

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

CLRS: Chapter 16 (Sections 1 and 2) and Chapter 23

DPV: Sections 5.1.1, 5.1.2, and 5.1.5

KT: Sections 4.1, 4.2, 4.3, and 4.5

SW: Section 4.3

# 17-1-Application - Internet Routing

The designer analysis of algorithms is the interplay between on the one hand general principles and on the other hand its stantiations of those principles to solve specific problems. While there's no silver bullet in algorithm design, no one technique which solves every computational problem that's ever going to come up. There are general design principles which have proven useful over and over again over the decades for solving problems that arise in different application domains. Those, of course, are the principles that we focus on in this class. For example, in part one we studied the divide and conquer algorithm design paradigm, principles of graph search amongst others. On the other hand, we study specific substantiations of these techniques. So in part one, we studied divide and conquer and how it applies to say, Strassen matrix multiplication, merge short and quicksort. In graph search, we culminated with the rightfully famous Dijkstra's algorithm for computing shortest paths. This, of course, is useful not just, because as any card-carrying computer scientist or programmer, you want to know about what these algorithms are and what they do, but it also gives us a toolbox, a suite of four free primitives, which we can apply to our own computational problems as a building block in some larger program. Part two of the course will continue this narrative. We'll learn very general algorithm paradigms. Like greedy algorithms, dynamic programming algorithms and many applications, including a number of algorithms for the greatest hits compilation. And in this video and the next, I want to whet your appetite for what's to come, by plucking out two of the applications that we'll study in detail later in the course. Specifically, in the dynamic programming section of the course. First of all, for both of these problems, I think their importance is self evident. I don't think I'll have to really discuss why these are interesting problems. Why, in some sense, we really need to solve these two problems. Secondly, these are quite tricky computational problems. And I would expect that most of you do not currently know good algorithms for these problems and it would be challenging to design one. Third, by the end of this class you will know efficient algorithms for both of these problems. In fact, you'll know something much better. You'll know general algorithm design techniques which solve as a special case these two problems and have the potential to solve problems coming up in your own projects as well. And one comment before we get started on these two videos. They're both at a higher level than most of the class, by which I mean there won't be any equations or math. There won't be any concrete pseudo-code, and I'll be glossing over lots of details. The point is just to convey the spirit of what we're going to be studying, and to illustrate the range of applications of the techniques that we're going to learn. So what I want to talk about first is distributed shortest path routing and why it is fundamental to how the internet works. So let me begin with a kind of non very mathematical claim. I claim that we can usefully think of the internet as a graph, as a collection of verticies and a collection of edges. So this is clearly an, clearly an ambiguous statement. There's many things I might mean as we'll discuss. But here's the primary interpretation I want you to have for this particular video. So to specify this, the vertices I intend to be the end-hosts and the routers of the internet. So machines that generate traffic, machines that consume traffic, and machines that help traffic get from one place to another. So the edges are going to be directed and they are meant to represent physical or wireless connections indicating that one machine can talk directly to another one by either a physical link between the two or a direct wireless connection. So it's common that you'll have edges in both directions, so that if machine A can talk to machine B directly, then also machine B can talk directly to machine A, but you definitely want to allow the possibility of asymmetric communication. So, for example, imagine I send an email from my Stanford account to one of my old mentors at Cornell, where I did my graduate studies. So this piece of data, this email, has to somehow migrate from my machine local at Stanford to my mentor's machine over at Cornell. So how does that actually happen? Well, initially there's a phase of, sort of local transportation, so, this piece of data has to get from my local machine to a place within the Stanford network that can talk to the rest of the world. Just like if I was trying to travel to Cornell, I would have to first use local transportation to get to San Francisco airport and only from there could I take an airplane. So this machine from which data can escape from the Stanford network to the outside world is called the gateway router. The Stanford gateway router passes it on to a networks, whose job is to cross the country. So last I checked, the commercial internet service provider of Stanford was Cogent so they, of course, have their own gateway router which can talk to the Stanford one and vice versa. And of course, these two nodes and the edges between them are just this tiny, tiny, tiny piece embedded in this massive graph, comprising all the end hosts and routers of the internet. So that's the main version of a graph that we're going to talk about in this video, but let me just pause to mention a couple of other graphs that are related to the internet, and quite interesting in their own right. So one graph that is generated an enormous amount of an interest in study is the graph induces by the web. So here, the vertices are going to represent web pages and the edges which is certainly directed represent hyperlinks. Not one web page points to another one. So for example, my homepage is one node in this massive, massive graph. And as you might expect, there is a link from my home page to the course page for this class. It is of course essential to use directed edges to faithfully model the web. There is for example, no directed edge from this courses homepage to my own homepage at Stanford. So the web really exploded around, you know, mid 90's, late 90's, So for the past 15 plus years, there's been lots of research, about the web graph. I'm sure you won't be surprised to hear that, you know, around the middle of the last decade, people got extremely excited about properties of social networks. Those, of course, can also be fruitfully thought of as graphs. Here, the vertices are going to be people, and the lengths are going to denote relationships. So, for example, friend relationships and Facebook or the following relationship on Twitter. So notice the different social networks may correspond to undirected or directed graphs. Facebook for example corresponding to an undirected graph, Twitter corresponding to a directed graph. So let's now return to the first interpretation I wanted to focus on, that where the vertices are in-hosts and routers and it does represent direct physical or wireless connections indicating that two machines can talk directly to each other. So going back to that graph, let's go back to the story where I'm sending an email to somebody at Cornell. And this data has to travel from my local machine to some local machine at Cornell. So, in particular, this piece of data has to get from the Stanford gateway router, in effect to the airport for Stanford's network to the Cornell gateway router. So there will be landing airport over on Cornell's side. Now it's not easy to figure out exactly out what the structure of the routes between Stanford and Cornell look like. But one thing I can promise you is that there is not a direct physical link between the Stanford gateway router and the Cornell gateway router. Any route between the two is going to comprise multiple hops. It will have intermediate stops. And there's not going to be a unique such route. So if you have the choice between taking one route which stops in Houston and then Atlanta and then in Washington D.C., how would you compare that to one which stops in Salt Lake City and Chicago. Well hopefully your first instinct and a perfectly good idea is all else being equal, prefer the path that is in some sense the shortest. Now in this context, shortest could mean many things, and it's interesting to think about different definitions. But for simplicity let's just focus on the fewest number of hops, equivalently the fewest number of intermediate stops. Well, if we want to actually execute this idea, we clearly need an algorithm that given a source and destination computes the shortest path between the two. So hopefully you feel well equipped to discuss this problem, because one of the highlights of part one of this class, was the discussion of Dijkstra's shortest pathalum rhythm in a blazing fast of limitation using heaps that run's in almost linear time. We did mention one caveat when we discussed Dijkstra's algorithm mainly that it requires all edge lengths to be non negative but in the context of internet routing almost any edge metric you'd imagine using will satisfy this non negativity assumption. There is, however, a serious issue with trying to apply Dijkstra's shortest path algorithm off the shelf to solve this distributed internet routing problem, and the issue was caused by the just massive, distributed scale of modern day internet. Probably back in the 1960's, when you had the 12-note ARPANET, you could get away with running Dijkstra's shortest path algorithm, but not in the twenty-first century. It's not feasible for the Stanford gateway router to maintain locally a reasonably accurate model of the entire internet graph. So how can we elude this issue? Is it fundamental that because the internet is so massive, it's impossible to run any shortest path algorithm? Well, the ray of hope would be if we could have a shortest path algorithm that admitted a distributed implementation. Whereby, a node could just interact, perhaps iteratively, with its neighbors with the machines to which its directly connected. And yet, somehow converge to having accurate shortest paths to all of the destinations. So perhaps, the first thing you'd try would be to seek out an implementation of Dijkstra's algorithm, where each vertex uses only local computation. That seems hard to do. If you look at the pseudo-code of Dijkstra, it just doesn't seem like a localizable algorithm. So instead, what we're going to do is we're going to learn a different shortest path algorithm. It's also a classic. Definitely on the greatest hits compilation. It's called the Bellman-Ford Algorithm. So the Bellman-Ford algorithm, as you'll see, can be thought of as a dynamic programming algorithm. And indeed, it correctly computes shortest path using only local computation. Each vertex only communicates in rounds with the other vertices to which it's directly connected. As a bonus, we'll see this algorithm also handles negative edge lengths. Which of course, Dijkstra's algorithm was not. But don't think Dijkstra's algorithm is obsolete. It still has faster running time in situations where you can get away with centralized computation. Now, it was really kind of amazing here is that the Bellman-Ford algorithm, it dates back to the 1950's. So, that's not just pre-internet, that pre-ARPANET. So that's before the internet was even a glimmer in anybody's eye. And yet, it really is the foundation for modern internet routing protocol's. Needless to say, there's a lot of really hard engineering work and further ideas required to translate the concept from Bellman-Ford to actually doing routing in the very complex modern day internet. But yet, those protocol's at their foundation, goes all the way back to the Bellman-Ford algorithm.

# 17-2-Application - Sequence Alignment

In this video we'll cover a second problem to whet your appetite for things to come, namely the problem of sequence alignment. So this is a fundamental problem in computational genomics. If you take a class on the subject it's very likely to occupy the very first couple of lectures. So in this problem you're given two strings over an alphabet and no prizes for guessing which is the alphabet we're most likely to care about. Typically, these strings represent portions of one or more genomes. And just as a toy running example you can just imagine that the two strings were given are A, G, G, G, C, T and A, G, G, C, A. Know that the two input strings do not necessarily need to be of the same length. And informally speaking, the goal of this sequence alignment problem is to figure out how similar the two input strings are. Obviously, I haven't told you what I mean by two strings being similar. That's something we'll develop over the next couple of slides. Why might you want to solve this problem? Well, there's actually a lot of reasons. Let me just give you two of many examples. What will be the conjecture or the function of regions of a genome that you don't understand, lets say the human genome, from similar regions that exist in genomes that you do understand or at least understand better, say the mouse genome. If you see a string that has a known function in the well understood genome and you see something similar in the poorly understood genome, you might conjecture it has the same or similar function. A totally different reason you might want to compare the genomes of two different species, is to figure out whether one evolved directly from the other and when. A second totally different reason you might want to compare the genomes of two different species is to understand their evolutionary relationship. So for example, maybe you have three species A, B, and C, and you're wondering whether B evolved from A and then C evolved from B, or whether B and C evolved independently from a common ancestor, A. And you might then take genome similarity as a measure of proximity in the evolutionary tree. So having motivated the informal version of the problem, let's work toward making it more formal. In particular, I owe you a discussion of what I mean by two strings being similar. So to develop intuition for this, let's revisit the two strings that we introduced on the previous slide A, G, G, G, C, T, and A, G, G, C, A. Now, if we just sort of eyeball these two strings, I mean clearly they're not the same string. But, we somehow feel like they're more similar than they are different. So, where does that intuition come from? Well, one way to make it more precise is to notice that these two strings can be nicely aligned in the following sense. Lets write down the longer string, A, G, G, G, C, T. And, I'm going to write the shorter string under it, and I'll insert a gap, a space to make the two strings have the same length. I'm going to put the space where there seems to be quote unquote a missing G. And then, what sense is this a nice alignment, well, it's clearly not perfect. We don't' get a character, by character match of the two strings, but there's only two minor flaws. So on the one hand, we did have to insert a gap and we do have to suffer one mismatch in the final column. So this institution motivates defining similarity between two strings with respect to their highest quality alignment, their nicest alignment. So we're getting closer to a formal problem statement, but it's still somewhat underdetermined. Specifically, we need to make precise why we might compare, why we might prefer one alignment over another. For example, is it better to have three gaps and no mismatches or is it better to have one gap and one mismatch? So if in this video, we're effectively going to punt on this question. We're going to assume this problem's already been solved experimentally, that it's known and provided this part of the input which is more costly, gaps and various types of mismatches. So here, then, is the formal problem statement. So, in addition to the two strings over A, C, G, T, we are provided as part of the input, a non-negative number indicating the cost we incurred in alignment for each gap that we insert. Similarly, for each possible mismatch of two characters, like, for example, mismatching an A and T. We're given as part of the input a corresponding penalty. Given this input, the responsibility of a sequence alignment algorithm is to output the alignment that minimizes the sum of the penalties. Another way to think of this output, the minimum penalty allignment is, we're trying to find in affect the minimum cost explanation for how one of these strings would've turned into the other. So we can think of a gap as sort of undoing a deletion that occurred some time in the past and we can think of a mismatch as representing a mutation. So this minimum possible total penalty, that is these values of this optimal alignment is famous and fundamental enough to have its own name namely the Needleman-Wunsch score. So this quantity is named after the two authors that proposed efficient algorithm for computing of the optimal alignment. that appeared way back in 1970, in the Journal of Molecular Biology. And now, at last, we have a formal definition of what it means for two strings to be similar. It means they have a small NW score, a score close to 0. So for example, if you have, if you have a database with a whole bunch of genome fragments, according to this, you're going to define the most similar fragments to be those with the smallest NW score. So, to bring the discussion back squarely into the land of algorithms, let me point out that this definition of genome sum, similarity is intrinsically algorithmic. This definition would be totally useless, unless there existed in efficient algorithm that given two strings and its penalties computes the best alignment between those two strings. If you couldn't compute the score, you would never use it as a measure of similarity. So this observation puts us under a lot of pressure to devise an efficient algorithm for finding the best alignment. So how are we going to do that? Well, we can always fall back to brute-force search, where we iterate over all of the conceivable alignments of the two strings, compute the total penalty of each of those alignments, and remember the best one. Clearly, correctness is not going to be an issue for brute-force search. It's correct essentially by definition. The issue is how long does it take? So let's ask a simpler question. Let's just think about, how many different alignments there are? How many possibilities do we have to try? So if [INAUDIBLE] let's imagine, I gave you two strings of length 500, which is a knot of a reasonable length. Which of the following english phrases best describes the number of possibilities, the number of alignments given to strings with 500 characters each? So I realize this is sort of a cheeky question, but I hope you can gather that what I was looking for was part D. So you know? So, how big are each of these quantities, anyways? Well, in a, in a typical version of this class, you might have about 50,000 students enrolled or so. So that's somewhere between 10^44 and 10^5.5. The number of people on earth is roughly 7,000.000.000. So that's somewhere between 10^9 and 10^10/10. The most common estimate I see for the number of atoms in the known universe is 10^80. And believe it or not, the number of possible alignments of two strings of length 500 is even bigger than that. So I'll leave it for you to convince yourself that the number of possibilities is at least two raised to the 500. the real number is actually noticeably bigger than that. and because 10 is at most 2^4, we can lower bound this number by 10^125 quite a bit bigger than the number of atoms in the universe. And the point of course, is just that it's utterly absurd to envision implementing brute-force search even at a scale of a few hundred characters. And you know, forgetting about these sort of astronomical, if you will, comparisons even if you had string lengths much smaller, say in the you know, a dozen or two, you'd never ever run brute-force or this is not going to work. And of course, notice this is not the kind of problem that's [INAUDIBLE] This just doesn't go away if you wait a little while for Moore's law to help you. This is a fundamental limitation. It says, you are never going to compute alignments of the strings that you care about, unless you have a fast, clever algorithm. . I'm happy to report that you will indeed learn such a fast and clever algorithm later on in this course. Even better, it's just going to be a straightforward instantiation of a much more general algorithm design paradigm. That of dynamic programming.

# 18-1-Introduction to Greedy Algorithms

Today, we're going to embark on the discussion of a new algorithm design paradigm. Namely, that of designing and analyzing greedy algorithms. So to put this study of greedy algorithms in a little bit of context, let's just zoom out. Let's both review some of the algorithm design paradigms that we've already seen, as well as look forward to some that we're going to learn later on, in this course. So it's sort of a sad fact of life that in algorithm design, there's no one silver bullet. There's no magic potion that's the cure for all your computational problems. So instead, the best we can do, and the focus of these courses, is to discuss general techniques that apply to lots of different problems that arise and lots of different domains. So that's what I mean by algorithm design paradigms. High level problem solving strategies that cut across multiple applications. So let's look at some examples. Back in part one, we started with the divide and conquer algorithm design paradigm. A canonical example of that paradigm being the merge sort algorithm. So remember in divide and conquer what you do is you take your problem you break it into smaller sub problems, you solve the sub problems recursively and then you combine the results into a solution for the original problem. Like how in merge sort you recursively sort two sub arrays and then merge the results to get a sorted version of the original input array. Another paradigm that we touched on in part one, although we didn't discuss it anywhere near as thoroughly, is that of randomized algorithms. So the idea that you could have code flip coins, that is, make random choices, inside the code itself. Often, this leads to simpler, more practical, or more elegant algorithms. A canonical application here is the quick sort algorithm using a random pivot element. But we also saw applications for example, to the design of hash functions. So the next measure paradigm we're going to discuss is that of greedy algorithms. So these are algorithms that iteratively make myopic decisions. In fact, we've already seen an example of a greedy algorithm in part one namely Dijkstra's shortest path algorithm. And then, the final paradigm we're going to discuss in this class is that of dynamic programming, a very powerful paradigm which solves, in particular, two of the motivating questions we saw earlier namely, sequence alignment, and distributed shortest paths. So what is a greedy algorithm anyways? Well, to be honest, I'm not going to offer you a formal definition. In fact, much blood and ink has been spilled over which algorithm is precisely greedy algorithms. But, I'll give you a sort of informal description. A sort of rule of thumb for what greedy algorithms usually look like. Generally speaking, what a greedy algorithm does, is make a sequence of decisions with each decision being made myopically. That is, it seems like a good idea at the time and then you hope that everything works out at the end. The best way to get a feel for greedy algorithms is to see examples and the upcoming lectures will give you a number of them. But I want to point out we've actually already seen an example of a greedy algorithm in part one of this course, namely Dijkstra's shortest path algorithm. So in what sense is Dijkstra's algorithm a greedy algorithm? Well if you recall the psuedo code for Dijkstra's algorithm, you'll recall there's one main wild loop and the algorithm process's exactly one new destination vertex in each iteration of this wild loop, so there's exactly N - 1 iterations overall, where N is the number of vertices. So the algorithm only gets one shot to compute the shortest path to a given destination. It never goes back and revisits the decision, in that sense the decisions are myoptic, irrevocable and that's the sense in which Dijkstra's algorithm is greedy. So let me pause for a moment to discuss the greedy algorithm design paradigm generally. Probably this discussion will seem a little abstract so I recommend you revisit this discussion on the slide after we've seen a few examples so at that point I think it will really hit home. So let me proceed by comparing it and contrasting it to the paradigm we've already studied in depth. That of divide and conquer algorithms. So you'll recall that in a divide and conquer algorithm what you do is, you break the problem into sub-problems. So, maybe you take an input array and you split it into two sub-arrays. Then you solve the smaller sub-problems recursively, and then you combine the results of the sub-problems into a solution to the original input. So the greedy paradigm is quite different in several respects. First, both a strength and a weakness of the greedy algorithm design paradigm is just how easy it is to apply. So it's often quite easy to come up with plausible greedy algorithms for a problem, even multiple difference plausible greedy algorithms. I think that a point of contrast with divide and conquer algorithms. Often it's tricky to come up with a plausible divide and conquer algorithm, and usually you have this eureka moment where you finally figure out how to decompose the problem in the right way. And once you have the eureka moment, you're good to go. So secondly, I'm happy to report that analyzing running time of greedy algorithms will generally be much easier than it was with divide and conquer algorithms. For divide and conquer algorithms it was really unclear whether they were fast or slow, because we had to understand the running time over multiple levels of recursion. On the one hand problems were size was getting smaller, but on the other hand, the number of some problems was proliferating. So we had to work hard, we developed these powerful tools like the master method, and some other techniques, for figuring out just how fast an algorithm like Merge Sort runs, or just how fast an algorithm like Strassen's fast matrix multiplication algorithm runs. In contrast with greedy algorithms, it will often be a one liner. Often it will be clear that the work is dominated by say, a sorting sub routine and of course we all know that sorting takes n log and time if you use a sensible algorithm for it. Now the catch, and this is the third point of comparison, is we're generally going to have to work much harder to understand correctness issues of greedy algorithms. For divide-and-conquer algorithms we didn't talk much about correctness. It was generally a pretty straightforward induction proof. You can review the lectures on Quicksort if you want an example of one of those canonical inductive correctness proofs. But the game totally changes with greedy algorithms. In fact, given a greedy algorithm we often won't even have very good intuition for whether or not they are correct. Let alone how to prove they're correct. So even with a correct algorithm, it's often hard to figure out, why it's correct. And in fact, if you remember only one thing from all of this greedy algorithm discussion many years from now, I hope one key thing you remember is they're often not correct. Often, especially if it's one you proposed yourself which you're very biased, in favor of. You will think the algorithm, the greedy algorithm must be correct because it's so natural. But many of them are not, so keep that in mind. So to give you some immediate practice with the ubiquitous incorrectness of natural algorithm. Let's review a point that we already covered in part one of this class concerning Dijkstra's algorithm. Now, in part one we made a big deal of what a justly famous algorithm Dijkstra's shortest path algorithm is, it runs brazenly fast and it computes all the shortest paths. What else do you want? Well remember there was an assumption when we proved that the Dijkstra's algorithm is correct. We assumed that every edge of the given network has a non negative length. We did not allow negative edge lengths. And as we discussed in part one, you know, for many applications, you only care about non negative edge lengths. But there are applications where you do want negative edge lengths. So let's review on this quiz why Dijkstra's is actually incorrect, despite being so natural. it's incorrect when edges can have negative lengths. So I've drawn in green, a very simple shortest path network with three edges and I've annotated the edges with their links. You'll notice one of those edges does have a negative length, the edge from V to W with length minus two. So the question is consider the source vertex S and the destination vertex W. And the question is, what is the shortest path distance computed by Dijkstra's algorithm and you may have to go and review just a pseudo code in part one or on the web. to answer that part of the question and then what is in fact the actual shortest path distance from S to W where as usual the length of a path is just the sum of the lengths of the edges in the path. All right, so the correct answer is D. So let's start with the second part of the question, what is the actual length of a shortest path from S to W when there's only two paths at all in the graph? The one straight from S to W that has length 2, and the one that goes by the intermediate point V that has length 3 + -21, = 1 which is shorter. So, SVW is the shortest path that has length 1. Why is Dijkstra incorrect? Well if you go back to the pseudo code of Dijkstra, you'll see that in the very first iteration it will greedily find the closest vertex to S in that case this is W, W is closer then V. It will greedily compute the shortest path distance to W knowing the information it has right now and all it knows is there's this one hot path from S to W, so it will irrevocably commute to the shortest path distance from S to W as 2. Never reconsidering that decision later. So Dijkstra will terminate with the incorrect output that the shortest path link from S to W is 2. This doesn't contradict anything we proved in part one, because we established correctness of Dijkstra only under the assumption that all edge links are non-negative, an assumption which is violated in this particular example. But again, the takeaway point here is that, you know, it's easy to write down a greedy algorithm, especially if you came up with it yourself. You probably believe deep in your heart that it's got to be correct all the time, but more often than not, probably your greedy heuristic is nothing more than a heuristic. And there will be instances in which it does the wrong thing. So keep that in mind in greedy algorithm design. So now that my conscience is clear, having warned you about the perils of greedy algorithm design, let's turn to proofs of correctness. That is if you have a greedy algorithm that is correct. And we will see some notable examples in the coming lectures. How would you actually establish that effect? Or if you have a greedy algorithm, and you don't know whether or not it is correct, how would you approach trying to understand which one it is, whether it's correct or not? So let me level with you. Proving greedy algorithm is correct. Frankly, is sort of, more art than science. So, unlike the divide and conquer paradigm, where everything was somewhat formulaic. We had these black box ways of evaluating recurrences. We had this sort of, template for proving algorithms correct. Really, proving correctness of greedy algorithms takes a lot of creativity. And it has a bit of an ad hoc flavor. That said, as usual, to the extent that they are recurring themes. That is what I will spend our time together emphasizing. So let me tell you just again about very high level. How you might go about this. You, again, might want to revisit this context aft-, content after you've seen some examples where I think it'll make a lot more sense. So method one is our old friend or perhaps nemesis depending on your disposition, namely proofs by induction. Now for a greedy algorithms remember what they do, they sequentially make a bunch of irrevocable decisions, so here the induction is going to be on decisions made by the algorithm. And if you go back to our proof of correctness of Dijkstra's algorithm, that in fact is exactly how we proved Dijkstra's algorithm correct. It was by induction of the number of iterations, in each iteration of the main wild loop. Computed the shortest path to one new destination. And we always proof that assuming all of our previous computations were correct, that's the inductive hypothesis. Then so is the computation in the current iteration. And so then by induction, everything the algorithm ever does is correct. So that's a greedy proof by induction that a greedy algorithm can be correct. And we might see some more examples of those in, for other algorithms in the lectures to come. Some of the text books call this method of proof greedy stays ahead, meaning you always proof greedy's doing the right thing iteration by iteration. So a second approach to approving the correctness of greedy algorithms which works in a lot of cases is what's called an exchange argument. So you haven't yet seen any examples of exchange arguments in this class so I can't really tell you what they are but that's what we're going to proceed next. I'm going to argue by an exchange argument that a couple of difference famous greedy algorithms are in fact corrected. It has a couple of different flavors one flavor is to approach it by contradiction. You assume for contradiction that a greedy algorithm is incorrect and then you show that you can take an optimal solution and exchange two elements of that optimal solution and get something even better which of course contradicts the assumption that you started with an optimal solution. a different flavor would be to gradually exchange an optimal solution into the one output by a greedy algorithm without making the solution any worse. That would show that the output of the greedy algorithm is in fact optimal. And formally that's done by an induction on the number of exchanges required to transfer an optimum solution into yours. And finally, I've already said it once, but let me say it again, there's not a whole lot of formula behind proving greedy algorithms correct, you often have to be quite creative, you might have to stitch together aspects of method one and method two, you might have to do something completely different. Really, any rigorous proof is fair game.

# 18-2-Application - Optimal Caching

So what are greedy algorithms good for? Well, it turns out they're well suited for a number of fundamental problems across different domains of computer science. And to wet your appetite, for the many examples that we're going to see, I want to begin by discussing the problem of optimal caching. The punchline of the lecture is going to be that a natural greedy algorithm in fact minimizes the number of cache misses over all possible ways of managing a small fast cache. So what is the caching problem? Well on the one hand, there's going to be a big, but slow memory which you can think of as holding everything you might be interested in. And then there's also going to be what we call a cache and so this is a much smaller memory to which access is much faster. So this situation comes up all the time across different doings, computer science, architecture, operating systems, networking. just to mention a couple of really obvious examples. You could imagine, the small fast memory being something like an L2 cache. And the big slow memory being main memory. Or perhaps actually main memory is the fast memory, and the big slow memory would be disc. Now, your task, in the caching problem is to process what we're going to call a sequence of page requests. So a page request just means that the client wants to access something in memory and it's guaranteed to be in the big slow memory. But if its not already in the small fast memory then you got to bring it in, you got to put it in there for the subsequent access. The algorithmic aspect of the problem answers the picture when there is a cache miss or also known as a page fault. That is when there is a request for some data which is not already in the cache. When that is the case, you have to bring it into the cache. The design question then is what do you evict from the cache in order to make room for this new piece of data which you have to bring in. So to illustrate the issues, let's look at extremely simple example. A cache that just has four slots for pieces of data. Let's assume that initially the cache is seated with the four pieces of data I'll call a, b, c, and d. Now remember the input is a sequence of page requests, so these are requests for different pieces of data. Now when a new request comes in, you're basically sitting there crossing your fingers that what's been requested is already in the cache. So for example, if the first request comes in for the piece of data marked c, you said good we're good to go it's already in the cache go ahead and access it. Similarly if the next request is for d, you don't have to do anything. Good times roll, and d just gets accessed directly. The problem arises when something is requested that's not in the cache. So let's say the next request is for the data item e. Now remember, you have to bring e into the cache and of course you have to evict one of these four pieces of data to make room for it, and your algorithm has to decide which. Can you get rid of a or b or c or d. For this example, let's assume that we evict a to make room for e. Assume further that the next request that comes in is for a new piece of data f. Again it's not in the cache so we have to evict something to make room for it. Let's assume we get rid of b in order to bring in f. And now an unhappy situation but something that could certainly occur is we get a request for something that used to be in the cache but which we have since evicted. So for example if the next request is for a, then we're stuck. It's going to be another page fault, we have to evict something to bring in a. And similarly, if there's a b, again we're paying the price for evicting b to make room for f in the past. So in this example with these particular choices for page evictions, we incur four page faults. Now the first two the e and the f there's nothing we could have done about it. We were given a cache that initially did not have e and f and then e and f showed up, well what are you going to do, you're going to miss no matter what. However, two were caused by our unfortunate eviction decisions early on, to evict a and b only to find them requested after eviction. And with 20/20 hindsight, we can conclude we really should have evicted c and d, not a and b, to make room for e and f. So the point of this example is to illustrate first of all the caching problem, how it works. You have this small fast memory, it can't contain everything at once and so you have to sort of manage the cache and evict things, to make room for stuff as it gets accessed. That's the first point of the example. The second point is to illustrate there's really two types of cache misses or page faults. There's the ones which you know, really you can't do anything about. No matter what algorithm you use, you're going to suffer those faults. But then, depending on the eviction algorithm, you maybe able to avoid some of the cache misses that you would incur with an inferior algorithm. Algorithm. So, now the obvious question is, how well can we do? What's the best algorithm? How do we minimize the number of cache misses, suffering only the ones that are inevitable? So this question was given a very elegant answer by Belady back in the 1960's. And I'm going to state the answer as a theorem. it's a theorem we're not going to prove, for reasons I'll discuss in a second. but what the theorem says is that a natural greedy algorithm is an optimal algorithm for the caching problem. That is, it minimizes the number of cache misses over any way you might think about managing the cache. And the natural greedy algorithm that is optimal, is called the furthest in the future algorithm. So what is the furthest in the future algorithm? Well, it's exactly what you think it would be. It's basically what seems like a good idea at the moment you have to perform a eviction from the cache. Basically you want to put off judgment day. You want to put off the regret of evicting this particular piece of data as long as possible. When are you going to regret evicting a piece of data? Well, it's when it gets requested next. So if we have four things in the cache, you know, one is one is requested next, one is requested in seven time steps and one is requested in you know, 70 time steps, that's the one you want to evict now because it will take the longest until you actually regret that eviction. So for example, in the example on the previous slide, you can check that the furthest in the future algorithm would in fact evict the ones you want to evict, a and b, not the ones that we evicted in the example, c and d. Now at this point, many of you are probably justifiably scratching your heads. You're wondering, you know, why is this useful. It doesn't seem like this is what we wanted. The objection, to this result being that the furthest in the future algorithm is clairvoyant. Its very definition assumes that you know the future, it assumes that at the moment that you have to make an eviction you're aware of when each of the pieces data in the cache will be requested next. But if you think for a minute about the motivating applications for sudding the ultimate caching problem, this assumption simply doesn't hold, you simply do not know the future, you simply do not know when each of the pieces of data in your cache will be requested next. So this algorithm is not defined, it is unimplementable. Despite that, this is still an extremely useful result to know. Why. Well, two reasons. First of all, this unimplementable algorithm can never the less, serve as a guide line for practical. Implementable algorithms. For example it naturally motivates the LRU or least recently used caching algorithm. So what you do in the LRU algorithm is that instead of looking forward in the future, which you can't do generally, you look in the past. And you say, well, let me guess that whatever's been requested recently, will be requested again soon. Whatever hasn't been request been requested for a long time, will continue to not be requested for a long time. So, that sets as a proxy for the piece of data that's going to be referenced the furthest down the future, you look for the one that was most recently referenced the furthest back in the past. So that's the LRU algorithm. And as long as data exhibits what's called locality of reference, meaning whatever's being requested a lot in the recent past is also going to be what's requested in the near future. Then LRU is going to approximate furthest in the future. And indeed, LRU is in many applications, the gold standard amongst practical implementable caching algorithms. The second reason this theorem is useful in practice is because it served as an idealized benchmark. A hypothetical perfect scenario against which you can compare your latest and greatest cashing hereistic. So for example, maybe you have a caching application and you start by implementing the LRU least recently used caching algorithm and then as a sanity check you probably want to go back later once you have hindsight you look at the last few days of traces of logs of page requests and you say, how well did we do. Let's look at how well our caching algorithm LRU did. And let's look at how well we would have done had we known the future. And hopefully, you're just a few percent away. And then you can conclude that, yes indeed, the data seem to have locality reference. Yes indeed, LRU is doing almost as well as if we know the future, and we can proceed. On the other hand, if you go back through the last few days of logs, and you find that your caching algorithm is doing much worse than furthest in the future, then it's back to the drawing board with respect to your caching algorithm. You should work harder, understand the data better, and come up with a smarter heuristic. So for almost all of the greedy algorithms that we cover in this course, I'm going to explain to you why they are correct. I'm going to prove it rigorously. this algorithm is an exception. I'm actually not going to prove this theorem for you. the way it works is by what's called an exchange argument. So again, you may not have seen any examples, but you will soon. but the exchange argument to prove the latter result, as far as I know it's pretty tricky, believe it or not. Even though the algorithm is natural, you might think this result feels a little self-evident. Try to prove it rigorously. Not easy. Not easy at all. Indeed if you look at a textbook and say operating systems or a field like that, generally you'll see a description of this algorithm. You'll see the claim that it's optimal but you won't find the proof. some algorithms textbooks, for example Algorithm Design by Kleinberg and Tardos, do include the proof of this theorem. Those of you that are interested, I challenge you to prove it yourself without looking it up on the web or in a textbook. I think if you try to prove it you'll appreciate the subtleties that come up in understanding whether greedy algorithms are correct or not. In the best case scenario, I would love. Love to see, a simple proof of this theorem, something that I could explain in a class like this, in say five minutes or less, that would be amazing.

# 19-1-Problem Definition

For our next case study of how to use greedy algorithms, we're going to turn to the application domain of scheduling. That is how do you schedule jobs on shared resources in order to accomplish some objective. So, the domain of scheduling there's lots of different applications of greedy algorithms. We'll see two in this course. we'll start for today just with the following simple scenario. So, we'll assume, for today, that there's just one shared resource. This resource could represent any number of things. For concreteness, you can think of it as a computer processor. And then, there's a lot of different things that got to get done. So, for example, there's a lot of processes that have to be handled by this processor. In the algprithmic question, we are going to study, is, in what order should we sequence these jobs? Which one should we do first, which one should we do second, and so on, all the way up to which one should we do last. So, obviously to answer this question, we need to pin down the mathematical model a little bit more precisely. And lets start with just, you know, what is the characteristics of jobs, what information do we have that might lead us to prefer one job over another. But for this problem, we're going to assume that each job comes with two known parameters. So, first of all, job j has what we're going to call a weight w sub j. That's a non-negative real number. And you should think of the weight of a job as quantifying its importance. That is, jobs with a higher weight, in some sense, deserve to be processed earlier than those with a lower weight. And secondly, each job j is going to come with a non negative length l sub j. Depending on the application, you may or may not have a good estimate of how long jobs are going to take, but for today to keep things simple, let's assume we know what the length of every job is, and that's l sub j, it's part of the input to are problem. So. we have now defined the input to this computational problem. We get n jobs each specified by a weight and a length. And we know that the output is going to be a sequence of these n jobs in some order. So, what we have to understand now is what criterion do we want to optimize? What are we trying to accomplish with this sequence? To explain that, I need to tell you about completion times of jobs. So, the completion time of a job is defined hopefully in exactly the way you'd think. So, for the job which is scheduled first, it's just the length of the job because that's how long it takes to process that job. For whatever job gets scheduled second, its completion time is the length of the first job and then, the length of that job itself. So, in other words, it's just the total time which elapses before that job gets completed, okay? So, in general, the completion time of a job is just the sum of the lengths of the jobs scheduled to before that job plus the length of that job itself. To make sure this is clear, let's go through a quick example. So, suppose there are three jobs with lengths one, two, and three. I'm not going to tell you the job weights because they're irrelevant for the purposes of computing the completion time. And let's suppose we do the schedule where we just schedule job one first, the job two, then job three. So, pictorially, I'm going to represent that schedule just by stacking the jobs on top of each other with the interpretation that time starts at the bottom. So, time zero is where we schedule job one. And then, time increases as we go from the bottom to the top of the diagram. And the question then is, what are the completion times of these three jobs? Okay. So, the correct answer is answer C. So, for the first job, it gets scheduled first so it's very happy and it just takes one unit of time to complete, so its completion time is one. The second job, well, it has to wait for the first job to complete so one unit of time elapses and then, it itself has to complete so that's two more units so it gets to the completion time of three. For the third job, it has to wait for the first two to complete, so that adds three to the clock, and then plus it takes three units of time for a total of six. So, that's the definition of job completion times. In some sense, we obviously want completion times to be as small as possible. But it's not so simple. In any given schedule, the jobs that are give early on are going to have small completion times and the jobs towards the end are going to have big completion times. So inevitably, we're going be have to trading off the completion times between different jobs. So, what is the optimal way to so that? Well, that depends on our objective function, and in scheduling, there's many different objective functions you might want to use. today, I'm just going to tell you about one. It's not the only natural objective function, but it's one of several most natural objective functions. It's called minimizing the weighted sum of completion times. You translate this English phrase into mathematics in the obvious way. What you want to do is you want to minimize the sum over all n jobs of their completion time, but then multiplied by their weight of [UNKNOWN] j. Okay. So, the sum over j of w j times c j. The w j is the weight and c j is the completion time as defined on the previous slot. If you think about it for a second, you'll realize this is equivalent to minimizing the weighted average of the completion times with the weights given as in the input. So, just to make sure this makes sense, let's go back to the example that we saw. In that example, we had jobs with lengths one, two, and three, and we thought about to schedule or we scheduled them in that order. To evaluate the subjective function, I'd have to tell you their weights, so let's suppose their weights are three, two, and one, respectively. In this case, the weighted sum of completion times in the schedule, in the previous slide, well first, we begin with a, the first job, which has weight three. Its completion time, remember, was one. Then, we have the second job with weight two, its completion time is three. Then, we have the third job with weight one, its completion time was six. So, we sum up the weighted completion times and we get a total of fifteen. And I'll let you verify that, in fact, all of the three factorial or six schedules in that example, this is, in fact, the schedule that minimizes the weighted sum of completion times. And the algorithmic question we're going to study next, is how do we do this in general? Given arbitrary input in jobs, weights, and lengths, what is the sequence that minimizes this sum over all n factorial sequences you might consider?

# 19-2-A Greedy Algorithm

So the plan for this video is to develop a greedy algorithm that always minimizes the waited sum of completion times of a given set of in jobs. But more than the specific problem, more than the specific algorithm I want you to focus on the process by which we arrive. At this greedy algorithm. because I think this process is really something which you can use yourself, in your own applications. The process we're going to use is we're going to first look at just a special case in the problem, where it's reasonably intuitive what should be the optimal thing to do. Looking at these special cases will then motivate a couple of natural greedy algorithms. Then, we'll figure out how to narrow a couple of greedy algorithms down to just a single candidate, and that in fact, as we will prove in the following lectures, will always be correct. So let's just briefly recall what is it we're trying to do. The computational problem and instant to specify by end jobs which come along with waits and links, and among all end factorial ways that we can sequence the jobs, we want to somehow home in on the one that minimizes the sum of waited completion times. Recall from the previous video that the completion time of a job is just the amount of time that elapses before it's done. So that's going to be the length of all the previous jobs plus the length of job J itself. What we're hoping is going to work out is that we can devise a greedy algorithm that always solves this problem. So maybe I should take a step back and ask, you know why do greedy algorithms seem like a sensible way to approach this scheduling problem? Well, you know. In general, greedy algorithms are not guaranteed to work. You may have to do something more complicated. But scheduling still seems like a good place to try them out. Remember what a greedy algorithm does, it iteratively makes myopic decisions. An then you hope you have a, a reasonably good result at the end. Now, what are we doing? We're studying a sequencing problem. The definition of the problem is to schedule a job then another job then another job all the way up to the last job and so this iterative nature of the solution suggests that at least if you're lucky if the problem is simple enough maybe there's a greedy algorithm which simply schedules the jobs in the correct order one at a time. So, we're going to see if that works for minimizing the sum of waited completion times. So let's start by thinking positive, being optimistic. So let's pause it that the greedy algorithm does exist for this problem. Given that we're in the greedy algortihm section of the course you're, probably you'll going to find this hard to believe. But suppose one existed, how would we discover what it is? Well a useful technique not just for this problem but, you know more generally in real life, first focus on some special cases of the problem, where it's relatively clear how you should proceed. And the two special cases I want you to think about for this problem are first of all, suppose I told you all of the jobs had exactly the same length but they had different weights, then, what order do you think it makes sense to schedule the jobs in? Secondly, suppose that I told you, that all of the jobs had exactly the same weight, but they had different lengths. Then, what order do you think you should schedule the jobs in? So first if all the jobs have the same length, you should prefer jobs with larger weights. Certainely, eh, this intuitively jives with our semantics of weights, that says more importance, which suggest that higher weight jobs should go first, if you look at the actual formula of minimizing the sum of weight and completion times, if the jobs all have the same length. Then the completion times are going to be the same your going to see the same set of them. If all the jobs have length one, then the completion times of the jobs are going to be one, two, three, four all the way up to N. No matter what sequence you use. So to make this as small as possible, you want the highest waits to be associated with the smallest completion times that is, you want them upfront as early as possible. The second special case where jobs have equal waits but varying lengths, I think is a little more subtle. Here what you want to do is you always want to favor small jobs, jobs with the smallest lengths, everything else being equal. The reason for that is that scheduling a job at a given position forces all the other jobs to wait for that job to complete. So all the, so whatever job you schedule first has a negative impact on all of the rest of the N minus one jobs. So I'll. Things being equal, you want the smallest job there that minimizes the consequences for the jobs that are to follow. If you find this a little unintuitive, I suggest just looking at a very simple example. Two jobs. Both have weight one, one has length one, one has length two. If you schedule the small job first you'll have completion times of one and three for a total of four but if you schedule the bigger job first you get completion times of two and three for the bigger sum of completion times of five. So the next step is to move beyond special cases, which we understand well. To the general case, which perhaps we don't understand. So suppose all of the weights are different, and all of the lengths are different. Well, if we have two jobs, and both of these rules of thumb give us the same advice, we're good. If there's one job which is both higher weight, and smaller than another job, then clearly that job should go first. But what if our two rules of thumb to prefer high weight jobs and to prefer small jobs, give us conflicting advice. What if we have a pair of jobs, where one of them is on the one hand higher weight, higher priority but on the other hand, bigger than the other one. Which one should go first? Well let's again stay positive, and let's try to think about the simplest kind of algorithm that could conceivably work. It won't be a guarantee that it works, but it might work. So we have these two different parameters, length and weights. Maybe we can aggregate these two parameters into a single one, into a single sort of score for each of the jobs so that if we schedule the jobs from high score to low score, we'll always be optimal. That would be great. If we could compile these two numbers into one for each job, and then just sort and be done. There is of course the question of exactly how do we choose this aggregation function. How do we compile length and weight into a single number. Well as guidelines we should recall our special case and make sure we respect our two rules of thumb. So all else being equal we should prefer jobs with higher weight. So that says higher weight should meet the higher scores if we're going to schedule the job from high score to low score. And then also if a length is bigger that should decrease the score. We should prefer jobs that have a small length. So this idea leaves open the question of exactly how do we aggregate the length and the weight of a job into a single number. So what I want you to do now is I want you to think for a minute about what kind of simplest possible functions you could use. So again, these are mathematical functions. They take as input two numbers, a length and a weight of a job, and they output a single number, a score. And the function should have the properties that it's increasing in the job's weight, and it's decreasing in the job's length. So there's more than one answer to this question but just sort of dream some sort of ideas of what this function might look like. Alright so there is certainly any number of functions that have these properties, but I'm just going to write down for concreteness two of what I think are of the simplest functions that have these properties. So one is going to be based on taking the difference of the two numbers and one is going to be based on taking the ratio of the two numbers. So if you're going to use a function based on the difference, then you're want to be increasing in the way of decreasing the length. Then of course, the obvious difference to use is weight minus length, this can be negative sometimes but that doesn't bother us the algorithm is still well defined. And if you're going to use a ratio and you want it to be increasing in weight, decreasing in length, then the sensible ratio to use is, the weight of a job, divided by the length of a job. It is of course possible that you have ties for either one of these scoring functions, so let's just allow ties to be broken arbitrarely. Now, what we're seeing here is a concrete instantiation of something I promised you in our high level discussion of greedy algorithms, namely, it's both a strength and a weakness of them that they're really easy to come up with and propose. So here we have just, you know, this one simple problem, and we now have two different, competing greedy algorithms for the problem. Now, because these two algorithms don't do the same thing. Only one of them, at most, can be always correct. At least one of them has to be wrong sometimes. So, as the algorithm designer, what the process now is. Maybe we can rule out at least one of these two proposed greedy algorithms, by showing an example where it doesn't do the right thing. So I want to emphasize this is the type of scenario that's very likely to arise in your own algorithm designed adventures. You might have some problem. You're not sure how to solve it yet. You've brainstormed up a couple of proposed algorithms. And a good thing to do, a good time saver is too quickly rule out some of those algorithms as not the right way to go as a poor approach to the problem. So in this context we have these two greedy algorithms. Let's quickly break one of them. Show that's it's not always correct. How do we do that? Well, a smart way to go would be to come up with an input where the two algorithms do different things. If they do different things, at most one of them is going to be correct. At least one of them is going to be incorrect so that's the plan. Now to execute this goal, as usual we want to keep things as simple as possible but no simpler. So, what's the simplest possible instance that could lead to different behavior by two algorithms? Well, obviously one job is not enough because there's only one possible feasible solution. But already with two jobs we might be able to have one algorithm flip them one way and the other algorithm schedule them in the opposite order. In fact it is not difficult to come up with an instance with two jobs where they do different things. let me just go ahead and write such an instance down for you now. So suppose I give you two jobs, the first one is both longer and more important than the other one, specifically, its length is five, its weight is three. The second job, its length is merely two but its weight is merely one. So what I want you to do is I want you to take our two proposed greedy algorithms, the first one which orders by difference, the second one which orders by ratio. I want you to execute them on this 2-job input and compute the sum of weighted completion times. And then answer what is the sum of weighted completion times of the corresponding two schedules. Alright, so the correct answer, is answer B. Let's just briefly go through why. So first let's just make sure we understand which algorithm produces which schedule. So the first job has the better ratio. It's ratio is five 3rds, where as the ratio of the second job is one-half which is smaller. Where as the second job has the larger difference, it has a difference of -1 where as the first job has the more negative difference of -2/ So, the first algorithm which orders by difference will schedule the second job first then the first job. The second algorithm will schedule the first job first and then the second one. So it just remains to compute the objective function value of those two schedules. So for the first schedule, with the second job first, while we're, the second job has waits of one, has a completion time of two. The second job has a weight of three and a completion time of seven. So that gives us a total of 23. Whereas the schedule produced by the second algorithm we have the weight three job first. It's completion time, now that it's first, is only five and then the second job with weight one get the completion time of seven for a total of 22. So ordering by difference gives us a value of 23. Ordering by ratio gives a value of 22. So in this case the ratio does better than the difference. So certainly the difference is not optimal for this specific example. So what have we accomplished? Well, what we've done is we very quickly ruled out one of our natural proposed greedy algorithms. We know that ordering by difference is not always correct. Again, it's going to be correct in special cases like when all the lengths are equal, where all weights are equal but it is not correct in general. That said, please remember the warning I gave you in the high level discussion of greedy algorithms which is greedy algorithms are very often wrong. Just because we know algorithm number one is incorrect sometimes does not at all imply that algorithm number two is guaranteed to be correct. It's really easy to come up with multiple incorrect greedy algorithms for the same problem. It does, however, turn out. In this case for this greedy algorithm, algorithm number two driven by ratio it is happily always correct. But you certainly shouldn't believe this claim until I provide you with a proof. A rigorous argument explaining the correctness. Always maintain healthy skepticism about the performance of a greedy algorithm until you learn otherwise. So, this fulfills another promise I gave you in the high-level discussion of greedy algorithms. Namely, when they're correct it's often quite difficult to prove it. So this would be the topic of the next couple videos the correctness proof for this greedy heuristic. The third and final thing we discussed about greedy algorithms typically is that their running time is not difficult to analyze. So that's a break that we catch relative to divide and conquer algorithms, and again that's certainly true here, right? So, what does this algorithm do? All it does is compute these ratios and then sort the jobs by ratio, so essentially the algorithm reduces to a single sorting computation and of course from part one, we know very well, how to sort in N log N time.

# 19-3-Correctness Proof - Part I

So now let's turn our attention to proving the correctness of this Greedy Algorithm that we devised that proportatedly minimizes the some of the waited completion times. Let me remind you of the formal correctness claim. The claim is that our second greedy algorithm. The one which looks at the ratio of each job. The ratio of the weight to the length and sorts them in decreasing order, is always correct. For every possible input, it ouputs the, the sequence of jobs, which minimizes the weighted sum of completion times. And as promised, it wasn't hard to devise the greedy algorithm. It's certainly not hard to analyze its running time, which is N Log N. The same time as sorting, but it is quite tricky to prove it correct. The way we're going to do that, is going to be our first example of what's called an exchange argument, which is one of the few recurring principles in the correctness proofs of greedy algorithms. So I'm actually going to give you proofs of two different versions of this claim. Both will make use of an exchange argument in slightly different ways. For starters that's going to be in the next video and the next one. I'm going to make a simplifying assumption that there are no ties amongst the ratios, that each job has its own distinct ratio of its weight versus its length. In this case, we will be able to get away with proof by contradiction. So, on this slide lemme give you the high level proof plan and of how this is going to go, we will start delving into the details on the next slide. So on the high level we're going to begin by fixing an arbitrary instance. By which I mean, you know just the description of the weights and lengths that. So remember, we have to prove that our algorithm is always correct. So we just fix an arbitrary instance, and prove correctness on this arbitrary instance. So, as I said, for the case with no ties, we're going to proceed by contradictions. Remember, this means we assume what we're trying to prove is false. And from that, we derive something which is obviously false inconsistent. So, what would it mean to assume that this claim is false? It means there exists an instances for which this greedy algorithm does not produce an optimal solution. For which there's some other solution not output by the greedy algorithm, which is better than that of the greedy algorithm. So let me just give you some notation to set this up. We're going to let sigma denote the greedy schedule. And if our claim is false, that means this is not an optimal schedule there's some other one which is better so call this better optimal sigma star. So, to complete a proof by contradiction, we need to derive something which is obviously false and the way we're going to do that here might strike you as initially as a little weird but it turns out to work really well in this context. From this assumptionm that the greedy algorithm is not optimal, and there is a better scheduled sigma star, we're actually going to exhibit yet another schedule which is even better than sigma star. Strictly smaller picto.function value than sigma star has. Why is that a contradiction? Well, by assumption, sigma star is optimal so if you show that there is something even better than sigma star, sigma star is not optimal and that completes the proof by contradiction. So now lets start filling the details of this proof plan and making it rigorous. So as I said in this video and the next we're going to be assuming that all of the ratios are distinct. In general, of course that need not be true and I'll give you a separate argument to handle the case of ties. I'm going to make a second assumption. But, unlike the first assumption, the second assumption has no content. It's just an assumption about notation. I'm going to assume by renaming jobs, that job one, number one, is the one with the highest ratio. Job number two is the one with the second highest ration and so on. Job ending the one with smallest ratio. As a consequence of this switch in notation the greeter schedule is very simple to describe. It just schedules job one first, then job two second, then job three third, and so on, all the way up to job N. Okay, so we have one assumption which is not without loss of generality, and we'll have a separate argument for handling ties. We have a second assumption which is without loss of generality. It's just an agreement amongst friends who want to minimize notation. And now, lets actually derive something with content. So given that the greedy schedule is just the jobs in order, one, two, three, all the way up to N. And given our assumption that the greedy solution is not optimal, and instead there's some other distinct optimal schedule sigma star. I claim that sigma star must contain consecutive jobs, that it is somewhere in the schedule, sigma star, I can isolate a pair of jobs, one executed after the other, such that the earlier of those two consecutive jobs has a larger index. I'm going to call these jobs I and J with I being earlier. So again by virtue路聽of the optimal solution sigma star being something other than the schedule one, two, three up to N there must be two jobs somewhere in the schedule executed in a row one after the other so that the earlier job I has a higher index then the subsequent job J. Why is this true? Well the reasoning is that, the only schedule. It has the property that indices only go up as you go from the earliest job to the latest job. The only way that indices will always go up is if you schedule the jobs one, two, three, all the way up to N. There is no other schedule with a property that indices always go up other than one, two, three, all the way up to N. So this is an observation that's going to be important in the rest of the proof, so make sure you pause, give yourself enough time to stare at it and commit yourself it is in fact true. Any, any schedule other than one, two, three, all the way through N, have to have consecutive jobs, the earlier one having a higher index than the later one. So I'm now in a position to explain the exchange in the exchange argument. So let me just distill the two key points from the discussion so far. So first of all we have changed notations so that ratios are decreasing with index and this is exactly the same as the schedule that the greedy algorithm will output. And then assuming that the optimal schedule sigma star is something else we know it has consecutive jobs with a earlier one having a higher index. Keep in mind, our high level proof plan from the first slide of this video. Where, we're doing a proof by contradiction. We need to derive a contradiction and what we're going to do is we're going to exhibit a schedule even better than sigma star thereby contradicting it's purported ophthalmology. So how do we do that? We do that with an exchange. So this exchange is going to take the place of methodic experiment. We're going to take this purportedly optimal schedule sigma star and we're going to switch the order just of the two jobs I and J leaving all of the other jobs unchanged. So sigma star consists of various jobs. It's called stuff collectively. then next is job I and after that immediately is job J. And then there's possibly some more jobs that get executed after J. And remember, we observe that, we can chose I and J so that I has a higher index than J despite being scheduled earlier. Then we execute this exchange. The stuff before INJ is the same as before. The more stuff after JNI, is the same as before. But we're going to have them occur in opposite order. And the key thing we have to understand next is, what are the ramifications of this exchange? What are the costs? What are the benefits? That's how we'll begin, the next video .

# 19-4-Correctness Proof - Part II

Okay let's continue the perfect correctness of our greedy algorithm for minimizing the sum of the weight of completion times and let's move onto understanding the ramifications of the exchange of jobs suggested at the conclusion of the previous video. So recall the basic ideas to use this observation that the optimal schedule sigma star by virtue of our assumption of being different from the greedy one has to have this pair of consecutive jobs where the earlier one has the higher index. So my question for you involves thinking about how the completion times of all of the jobs change after we make this exchange of the two jobs i and j. Which ones go up? Which ones go down? Which ones are unaffected? Which ones can we not actually predict whether they go up or down? All right, so the answer to this, quite key question is the third answer. Jobs other than I and J are unaffected, the completion time of job I goes up, and the completion time of job J goes down. So let's review why. Consider a job K, other than I or J, it's probably easiest to see if in sigma-star, K completes before I and J, scheduled earlier than I and J. Don't, remember what the completion time of a job is, it's just the time that needs to elapse before it gets done, so it's the sum of the job lengths up to, and including that job itself. So if K was scheduled before I and J before, it's exactly the same after I and J are swapped. You don't know the difference. Exactly the same set of jobs precedes Job K is the force whose completion time is the same. But if you think about it, this exact same argument is true for jobs K that succeed INJ. So before we do the swap, it has to wait for a bunches ops to complete, including INJ and after we swap INJ, it still has to wait for INJ to complete. Yeah, they complete in opposite order but who cares? The amount of time of the lapse is exactly the same. So importantly, jobs other than INJ are completely agnostic to the swap. Their completion time is exactly the same as before. So that's the first. Part. so job I, it's completion time goes up. It's easier to see why it used to be before J, now it has to wait for J. In fact we can say exactly how much completion time goes up by. It goes up by exactly the length of J. That is the extra time that now needs to elapse before I gets completed. By exactly the same reasoning the completion time of job j actually drops it has to wait for the same jobs to complete before it being accepted no longer has to wait for job I. So, not only can we say its completion time goes down we can say that it goes down so by precisely the length of job I. So now we are in a great position to finish off this proof. Let's summarize what we got so far. So, for a cost benefit analysis of exchanging I and J, we discovered the cost is limited to an increase in the completion time of job I and the benefit is limited to the decrease in the completion time of job J. Specifically the cost, the new cost incurred by the exchange is the weight of job I times the amount by which it's completion time went up, namely the length of job J. Similarly, the benefit of the exchange is the drop of LI and J's completion time and that gets multiplied by it's weight's of WJ. So now finally we are at the point where we can use the fact that this purportedly optimal schedule sigma star schedules I and J in some sense incorrectly, with a higher index job first despite it having a lower ratio in contrast to the principles followed by our greedy algorithm. So why is it relevant that the earlier job I has a higher index? Well, a higher index corresponds to a lower ratio. Remember we did this notation switch so that the first job has the highest ratio, the second job has the second highest ratio and so on. So the bigger your index, the further you are down in the ordering, the lower your ratio. So because I is bugger than J, that means I's ratio is worse than J. The usefulness of this becomes even more apparent when we clear denominators. We multiply both sides by the, by LI times LJ. And then we find that WI times LJ is strictly less than WJ times LI. But what are these, these are just the cost and benefit terms respectively of our thought experiment exchange, exchanging I and j. So what does it mean that the benefit of the swap outweighs the cost? It mean if we take sigma star, this reportedly optimal solution, and we invert the jobs I and j, we get a schedule with an even better weighted sum of completion times, but that is nuts. That's absurd, sigma star was supposed to be optimal. So here's our contradiction. That completes the proof.

# 19-5-Handling Ties [Advanced - Optional]

So in this video, we're going to revisit our greedy algorithm for minimizing the wait and somewhat completion times. And we're going to give a more robust more general correctness proof that also accommodates ties amongst the ratios of the different jobs. The main reason I'm doing this is not because you know, I think the result is, is so earth-shattering in its own right, but rather to give you further examples of exchange arguments in action, in particular outside of a proof by contradiction. So let's be formal about our new more general correctness proof. So we're again talking about the greedy algorithm which orders jobs by the ratio of weight to the length. We're no longer assuming these are distinct. So we can't really say decreasing order. We'll say non-increasing order. And ties can be broken any way you want. We'll prove the algorithms commit, correct, no matter how the ties are resolved. So fortunately,, we'll be able to reuse much of the work that we did for the previous correctness proof. But actually, our overall proof plan is going to change a little bit. we're no longer going to proceed by contradiction. So here's the high level, here's the high level plan of that. So as before, we're going to argue correctness on every separate instance, so fix an arbitrary one. So, the notation will be similar to last time, so on this input, we'll let sigma denote the output of our greedy algorithm and then we'll let sigma star denote an arbitrary other competitor, any other schedule of the jobs. So now, we're going to do is were going to show that sigma, the output of our greedy algorithm, is at least as good as sigma star, since sigma star was arbitrary, that means the greedy algorithms output is at least as good as every other schedule, and therefore, sigma has to be optimal. So, let's now fill in the details. We're going to make a similar notational assumption as last time and that we're going to assume that the greedy schedule is just 1, 2, 3, all the way up to n. And again, that's a content free assumption, we can get away with it just by changing the names of the jobs, changing notation. So, recall the proof plan, we have to take any other competing schedule sigma star and show that it's no better than sigma, show that sigma is at least as good as it. So fix any such schedule sigma star, obviously, if sigma star is sigma, then they're they same or just as good as each other so there's nothing to do. Now, if sigma star is not the same thing as sigma, if it's not just the sequence 1, 2, 3, all the way up to n, we're going to reuse a key observation from our previous proof, namely any schedule other than just 1 through n has to contain in it a consecutive pair of jobs, i and j, where j is executed immediately after i where i has the higher index. So now we argue similarly to last time. What does it matter that's, one job has a higher index than another. Well, that means it's further along in the ordering, which means its ratio can only be smaller. Remember, the ratios are non-increasing, they can only go down in the order. So higher index, means lower ratio. But there may be ties, so we can't claim a strict inequality, just a weak inequality. And so again, by clearing denominators, this boils down to the weight of job i times the length of job j is at most the weight of job j times the length of job i. Now, the next thing I want you to recall from our previous proof is that there are nice semantics with wilj and wjli namely as the cost in the benefit of exchanging jobs i and j in the schedule sigma star. So arguing as in the previous videos, we can argue, we can claim that exchanging i and j [SOUND] from a schedule sigma star has a net benefit, that is a benefit minus cost of wjli, that's because job j's completion time drops by li minus wilj. That's because job i's completion time increases by lj with this exchange and so this is non-negative. [SOUND] So in comparison with our previous proof, in our previous proof, with the assumption of bought us, it bought us the stronger fact than we exchange i and j, in fact sigma star gets strictly better. It's we get a better schedule than what we started with. Here with ties, that need not be true. If i and j have exactly the same ratio and we exchange them, then the cost equals the benefit so the net change in the objective function is 0. So we can only claim that by inverting i and j, we don't make sigma star worse. It can only get better and might stay the same. So let's see why that's good enough to nonetheless complete the proof. So what the previous slide gives us, it gives us a method of changing a schedule, massaging a schedule, so that, it doesn't get any worse, it can only get better. Specifically, if we take any schedule sigma star, we take any adjacent inversion, by which I just mean two consecutive jobs with the higher one having a higher index. We exchange the jobs in any adjacent inversion, we get a new schedule which can only be better. Some of weighted completion times might be the same, but if it's different, it has to be smaller. So in our previous proof, we knew it was strictly better because we had no ties and then our proof by contradiction said we were done. So what are we going to do now? What we're going to do is take this operation, which massages the schedule without making it worse, and we're just going to repeat it over and over and over again, because actually, this operation has a second property. Not only can it not make a schedule worse, but, it also decreases the number of inverted job pairs. And here by an inversion, I mean the same thing as when we counted inversions with the divide and conquer algorithm back in part 1. I just need a job pair somewhere in the schedule, where the higher next to one occurs earlier in the ordering. When we exchange adjacent inversion, we uninvert that aversion and because they are adjacent, we don't create any new inversions. So the number of inverted job pairs drops by exactly one, each time we do one of these exchanges. So what does that mean? So here is the proof in a nutshell. We take an arbitrary competitor, some schedule sigma star and we find, either it's the same as the greedy schedule. If it's not, there is an adjacent inversion, in which case we exchange it. We get a schedule that's at least as good and fewer inversions. Either this new schedule is sigma, in which case we're done, or it's not, and then we find an adjacent inversion, and we exchange it, it only gets better and we keep going. Why can this not continue forever? Well, the number of inversions can only be n choose 2 initially, that's if you start with the schedule n, n - 1, n - 2, all the way 1 if the jobs are initially backward. So, we can only do this exchange n choose 2 times before we necessarily terminates witht the greedy schedule, 1 through n. At that point what have we done? We've taken an arbitrary schedule sigma star, we've massaged it, making it only better, and better, and better, and better, terminating with our greedy schedule sigma. What does that say? That says our greedy schedule sigma is at least as good as when we started with sigma star. So the greedy schedule is at least as good as this arbitrary schedule sigma star, so it's optimal. It's better than everything. So one final note before I write down the QED. for those of you familiar with the bubble sort algorithm and it's totally fine if you're not familar with bubble sort, but if you are familiar with bubble sort, you will recognize that essentially what we're doing here inside the proof, not in tour algorithm but inside our proof, we're applying bubble sort in effect to this arbitrary competing schedule sigma star. And by uninverting the inversion we transform it in to the, the greedy schedule, making it only better, thereby justifying as optimal our greedy algorithm schedule sigma.

# 20-1-MST Problem Definition

So in this sequence of videos, we're going to apply the greedy algorithm design paradigm to a fundamental graph problem, the problem of computing minimum spanning trees. The MST problem is a really fun playground for greedy algorithm design, because it's the singular problem in which pretty much any greedy algorithm you come up with seems to work. So we'll talk about a couple of the famous ones, show why they're correct, and show how they can be implemented using suitable data structures to be blazingly fast. So, I'll give you the formal problem definition on the next slide but first let me just say informally what it is we're trying to accomplish. Essentially, what we want do is connect a bunch of points together as cheaply as possible. And, as usual with an abstract problem the objects can mean something very literal. So maybe the points we're trying to connect are servers in some computer network, or it could represent something more abstract. Like maybe we have a model of documents like Web Pages where we represent them as points in space. And we want to somehow connect those together. Now the main reason I'm going to spend time on the minimum expenditure problem is pedagogical. It's just a great problem for sharpening your skills with greedy algoritum design and proof of correctness. It'll also give us another opportunity to see the beautiful interplay between data structures and fast limitation of graph algorithms. That said that minimum expenditure problem does have applications. One very cool one is in clustering, and that I'll talk about in detail in a later video, it also comes up in networking. So if you do a web search on spanning tree protocol you'll also find some information about that. So as I said at the beginning the minimum spanning tree problem is remarkable in that it doesn't just admit one greedy algorithm that's correct, but in fact it admits multiple greedy algorithms that are correct. we're going to talk about two of them, the two most well known ones. But there are even some others believe it or not. So the first one we're going to discuss beginning in the next video is Prim's MST algorithm. This dates back over 50 years to 1957. in fact as you'll see Prim's algorithm shows a remarkable number of similarities with Dijkstra's shortest path algorithm. So you might not be surprised to know that Dijkstra also independently had discovered this algorithm a couple of years later. But in fact it was only noticed much later that this exact same algorithm had been first discovered over 25 years earlier by a mathematician named Jarnick. For that reason you'll sometimes hear this called Jarnick's algorithm or the Prim-Jarnick algorithm. for gravity and to be consistent with some of the main text books in the area I'm just going to call this Prim's algorithm throughout the lectures. The other algorithm we're going to cover which is also rightfully famous is Kruskal's MST algorithm. As far as I know this was indeed first discovered by Kruskal roughly the same time as Prim was doing his algorithm in the mid 50s. And in what sense do I say these algorithms are blazingly fast? Well, they run in almost linear time, linear in the number of edges of the graph. Specifically we'll see how using appropriate data structures will get each of them to run in time big O of M log N, where M is the number of edges in the graph, and N is the number of vertices in the graph. We'll employ data structures to speed up Prim's algorithm in exactly the same way we did for Dijkstra's algorithm, that is we'll be using the heap data structure, One thing that's cool about Crystal's algorithm is it'll give us an opportunity to study a new data structure, mainly the union fine data structure and that's a lot of fun to think about, in its own right, as you'll see. So to put this amazing running time on perspective I want to emphasize that only is it awesome in the sense it's you know, barely, it's almost linear. It takes almost barely more time to compute the spanning tree than it does to read the input graph. Reading the input graph alone, remember would take linear time. O of M time. But more over, graphs can have an enormous number of spanning trees. An exponential number. So some of these algorithms are honing in really quickly on a needle in a haystack. There's no way they have time to look at all these spanning tees, and yet they find the one which is the best which is optimal amongst all of them. How do these seemingly magical algorithms do it? Well, to discuss the details let's start by formalizing the Minimum Spanning Tree, or MST problem on the next slot. So in the MSD problem this is a graph problem so the main part of the input is a graph comprising verticies and edges. I do want to emphasize for the MST problem we are be considering only undirected graphs. This is different notice, than when we discussed shortest-path problems in Part one of the course. There we worked with directed graphs. There is an analogous problem to the [INAUDIBLE] signature problem for directed graphs. It's often called the optimal branching problem. And there are fast algorithms for it, but those algorithms are just slightly beyond the scope of this course. So we're not going to cover it. We're going to discuss only undirected graphs, and then minimum spanning trees for them. Now, whenever you talk about graph problems, you need to talk about, how is the graph actually represented. So that's something we discussed at length in part one. If you don't remember, I suggest going back and reviewing the video on graph representations. For the MST problem, we're going to assume that the graph is given as an adjacency list. That means, we're given an array of vertices, an array of edges. And we have pointers, wiring vertices to their incident edges and wiring edges back to their two endpoints. In addition to the graph of self the input includes a cost, for each of the edges, we're going to use the notation C sebies of note the cost of a edge, E. And in another contrast, to are discussion of shortest path problems, we're actually not going to care if the edge cost are positive or negative, they can be any number whatsoever. So no prizes for guessing what the outputs supposed to be, it's right there in the problem definition, the output is supposed to be a minimum cost spanning tree of the graph, but let's drill down and explain exactly what we mean by that. So first of all what do we mean by the cost of a tree or generally the cost of a sub graph, as a subset of the edges. Well we're just going to be looking at summing up the edges in the tree that we output. Now the other question is what do I mean by a tree that spans all vertices? So let me tell you exactly what this means, the sub graph T should have two properties, first of all there can not be any cycles, there can not be any loops in this tree. And by spanning all vertices, what I mean is that this sub graph is what's called connected. That is, there's a path, using the edges and t, from any vertex of the graph to any other vertex. That's what it means to span all of the vertices. So for example, consider the following graph with four vertices and five edges. I've labeled each of the five edges with a cost, which in this case, is just an integer between one and five. So, let's look at some example subgraphs, let's start with the three edges, A, B, B, D and CD. This sub-graph satisfies properties one and two. That is, it has no cycles, there's no loops and it spans all of the vertices. If you start at any one of these four vertices, you can get to any of the other four vertices by using only red edges. So in that sense, this red sub-graph is a spanning tree. However, it is not the minimum cost spanning tree. There is another spanning tree which is even cheaper, has a smaller sum of edge costs, namely the edges AC, AB, and BD. This also has no cycles and it's also connected but the sum of the edge cost is only seven, smaller than the eight of the previous spanning tree. In fact, this pixograph is the unique minimum spanning tree of this graph. There is a sub graph that has three edges which has an even smaller sum, of edge costs, namely the triangle AB, BD and AD. But this light blue sub graph, this triangle, is not a spanning tree. In fact, it fails on both counts. It does obviously have a cycle. It has a loop. That's, what it is by definition. It's also not connected, so there's no way to get from C, the vertex, to any of the other three vertices by following only light blue edges. It's disconnected, and so it fails property one as well. So the MST problem in general is you're given it under a graph, like, for example, this four note, five edge graph, or presumably. something much larger and an interesting problem and your suppose to quickly identify the minimum spanding tree like in this example the pink subgraph. So what I want to do next is something you're probably quite accustomed to me doing by this point, is I want to make a couple of mild simplifying assumptions just among friends. So these assumptions are not important in the sense that all of the conclusions of these lectures will remain true, will remain valid even if these assumptions are violated but it'll make the lectures a little bit easier. It'll allow us to focus on the main points and not get distracted by less relevant details so here are the two assumptions that we're going to make throughout all of the lectures on minimum spanning trees. The first assumption we're going to make is that the input graph G is itself connected. That is G contains a path from any vertex to any other vertex. So why am I making this assumption? Well if this assumptions violated then the problem isn't even well defined. If the graph isn't connected then certainly none of it's subgraphs are connected so it has no spanning trees and it's not clear what we're trying to do. So, those of you who still remember the stuff we covered in part one in particular, graph search. Should recognize that this condition's easy to check in a pre-processing step. Just run something like breadth first search or depth first search. Remember, we know how to implement those in linear time. And those will, in particular, tell you whether or not the input graph is connected. Now, another thing you might be wondering is, suppose it was disconnected. Then what? Should be really just sort of throw up our hands and give up? You can define a version of the minimum spanning tree problem. A more general one called minimum spanning forest. Where, basically you want the minimum cost sub graph that spans as much stuff as possible. Essentially, it's responsible for computing a spanning tree within each of the connected components of the original graph. And using the algorithms I'll show you here, Prim's algorithm, Kruskal's algorithm, they're easily modified to solve the more general problem with disconnected input graphs as well. But again, for simplicity among friends, let's just focus on the connected graph case that contains all of the main ideas. Our second standing assumption throughout all of the minimum of spanning tree lectures will be that in the input graph the edge costs are distinct. So you're already use to this sort of no ties kind of assumption from our foray into scheduling algorithms, and we're going to do something similar here. Now again this assumption is not important in the sense that the algorithms that we cover prims algorithm crustgrals algorithm. They remain correct even if the input has equal cost edges, irrespective of how ties are broken. So the algorithms are correct as widely as you would want. That's it. I'm not going to actually prove for you that they are correct with ties. Remember we had our scheduling, application it was a little bit easier to get a proof of correctness without ties, I gave you that, and then optionally there was a slightly more complicated argument that handled ties. You can do the same thing here, but I'm just not going to give it to you. I'll leave that for the keen viewer to work out for themselves.

# 20-2-Prim's MST Algorithm

Okay, so it's time to discuss our first minimum spanning tree algorithm namely Prim's algorithm. Definitely a candidate for the greatest hits compilation. And again remember even though it's called Prim's algorithm, it was actually discovered earlier by Jarnik. So how's it work? Well before showing you any pseudo code, let's first illustrate it on an example. As we go through the example, I hope that the similarities to Dijkstra's shortest path algorithm will be evident. I'm going to work with the same example graph from the previous video with four vertices and five edges. The plan is to grow a tree one edge at a time. And we're going to keep growing this tree like a mold. We're going to start from just a seed vertex. And then we're going to suck up one new vertex with each iteration of the algorithm. So, this is similar to Dijkstra's Algorithm. In Dijkstra's Algorithm, it was clear where we should grow the initial mold from, because we were given a source vertex, that they're trying to compute the shortest paths out of. We have no source vertex in the minimum spanning tree problem, but it turns out that we can just pick an arbitrary vertex to start. Doesn't matter which one, which is cool. So the plan is in E generation we're going to add one edge and span one new vertex adjacent to the ones we're already spanning. Now as a greedy algorithm Prim is simply going to select the cheapest edge that allows it to span one additional new vertex. Now the start of the algorithm here we're not really spamming anything. We are sort of thinking of ourselves as growing from and currently spanning the vertex in the upper right. So what are the edges in which we can span an adjacent vertex? Well, there is two inches. There is the top inch that costs one then we'll span an addition in the upper left vertex or the is the edge with cost two on the right. If we include that, we'll be able to span the vertex in the bottom right. So we're not going to be greedy, we're just going to choose the cheaper edge, the edge of cost one. Now, the vertices that our tree thus far spans are the top two vertices. So, in the next iteration, we want to add one more edge [COUGH] to span one additional new vertex. And now we see that there are three edges sticking out of what we're spanning thus far that will allow us to span a new edge. There's the edges that have cost two, three, and four. The two and the three will allow us to span the vertex in the bottom right. If we pick the four, that will allow us to span the vertex in the bottom left. Yeah, and we're going to be greedy, so of these three candid edges, we're going to pick the cheapest one which is the edge of cost two. So now the mold that we've been growing is in effect, covers all of the verticies except for the one in the bottom left. So now in the final iteration we want to include one more edge so that we span that final remaining vertex. The one in the bottom left. Note that there's there was this edge of cause three that we never added. But it got sucked up into the tree that we grew anyways. So we're going to go ahead and ignore that. Adding the three wouldn't allow us to span any more vertices. In fact, it would create a loop which we don't want. So we're going to say, okay. We'll have the two edges that would allow us to span an extra vertex. There's the four and there's the five. We're going to be greedy, we're going to pick the four. And once we have the edges of the cost one and two and three and four we have a spanning tree there's no loops there's a path from any vertex to any other vertex along the pink edges, the cost is seven you might recall from the previous video this is indeed the minimum cost spanning tree of this graph. Of course, the fact that we have this simple procedure that works correctly in this toy example, which is four vertices and five edges, really means nothing. I mean you shouldn't draw any immediate conclusions that this is a good algorithm in general even though that is going to be the case. So let's next go and actually define the algorithm generally. So if you have a general graph, what does it mean to start somewhere and grow a mold, span one more vertex each iteration, always proceeding greedily until you are done. So lets spell out the pseudo code on the next slide. So here is Prim's minimum spanning tree algorithm. We're going to start with just two lines of initialization. We're going to maintain a set of vertices, capital X. This is meant to the be the vertices that we span so far. Again, we need some seed vertex from which to start the process. It doesn't matter where, which one we pick. We're going to get the same tree no matter what, so just call it little s. That's an arbitrary vertex from which we start growth. The other thing we're maintaining is, of course, the tree. So that's initially going to be empty. We're going to add one edge to it in each iteration. An invarient that we are going to maintain throughout the algorithm is that the edges that currently reside in the set capital T span the verticies that currently reside in the set capital X. Then we're going to have our main while loop. this is the workhorse of the algorithm. And it's very similar to the one in Dijkstra's algorithm. Namely, each iteration is responsible for picking one edge crossing the current frontier. advancing to include one new vertex. And again, it's going to be greed. The criterion's going to be different, in fact, simpler, than with Dijkstra's Algorithm instead of looking at links. We're just going to say, what's the cheapest edge that allows us to span a new vertex? So the loop's going to keep going, as long as there are vertices that we don't yet span. And then what we do is we search to the edges that allow us to span a new vertex. So which edges are those? Well we want there to be one endpoint in the set X of vertices we already have our tree spanning and we want the other end point to be non-redundant, so we want it to be outside of X. So if we have an edge that crosses the frontier in this sense, one endpoint in X, in endpoint outside that's how we increase the number of spanned vertices by one in an iteration. So if E is the cheapest edge amongst all of those that cross the front here with one end point on either side, that's the one we're going to add to our tree so far capital T in this iteration, it's end point that's not already in capital X, that's going to be the very text that we add to X in this iteration. And again the semantics of an iteration is that we're trying to increase the number of spanned vertices while paying as little as possible, that's the sense in which a prim's algorithm is a greedy algorithm. So as usual with a greedy algorithm, this seems natural enough, but it's not at all clear that it's correct, that it always computes in minimum spanning tree. In fact, if you think about it's not even obvious, it actually computes a spannin tree at all, minimum or otherwise, but it is correct. Let's make that statement precise on the next slide. So the key claim is that Prim's Algorithm is correct. Given any connected input graph, it is guaranteed to output a spanning tree with minimum possible cost. So before we delve into any details, let me just finish this video by telling you about the proof plan. We're going to prove this theorem in two parts. First, we're going to establish that it outputs some spanning tree. Maybe, maybe not minimum. Even that's non trivial. Then we'll worry about arguing that the spanning tree output actually is one of minimum cost. Both parts of the proof are interesting. For part one to argue that we output some spanning tree, we're going to review some preliminaries about graphs and about cuts and about spanning trees and graphs. For part two to argue optimality, we're going to rely on a very neat property of spanning trees, minimum spanning trees called the cut property. I'm happy to report so that the work that we do here and in both parts will bear further fruit later we're going to reuse these ingredients when we prove the correctness of another MST algorithm named McCrustgrals algorithm. For those of you who would much rather talk about running time than correctness don't worry your time will come after we wrap up this correctness proof I'll address how do you implement prim's algorithm quickly in particular using heaps we'll get the running time down to the near linear bound of O of M log n.

# 20-3-Correctness Proof I

Okay. So in this video we're going to begin our discussion about why Prim's algorithm is correct. Why always, for every connected graph outputs a minimum spanning tree of that graph. For this video, we're going to content ourselves with a much more modest school. We're only going to prove for now the Prim's algorithm outputs a spanning tree. We're not going to make any claims yet about optimality. Even just this fact is not trivial and proving it will give us a good opportunity to get our hands dirty with some basic properties of graphs and specifically graph cuts. Graduates of part 1 of this online class of course are already familiar with graph cuts. We studied them at length via Karger's randomized algorithm for computing the minimum cut of a graph. So, the concept is the same here, let me state it again to jog your memory. So a cut of a graph is simply a partition of its vertex set, two groups, and each of those two groups should be non-empty. So pictorially, we envision some of the vertices of G, this blob A being in one group, and the rest of the vertices, this graph B being in a different group. Now, what's up with the edges? How can they be distributed in this picture? Well, the two endpoints of an edge, there's three cases, either both of the endpoints can be in the set A. So there's various edges internal to A. Similarly, an edge might have both of its endpoints inside of B. But we're going to be most interested in the third case, edges that have one point exactly in each of A and B. So these are edges that we say cross the cut, A, B. So hopefully the definition of a cut seems simple enough, but cuts in particular their relationship to edges can be quite interesting, quite useful. So as shown here in the picture, of course for a given cut, there can be many edges crossing it. by the same token for a given edge of a graph, in general, there will be many cuts of the grap, that's, that edge crosses. So, to understand this a little bit better, let's just review a simple property that cuts through the graph. Let me just ask you just how many there are. Specifically, for a graph that has n vertices, roughly how many cuts does it have? Roughly n, roughly n squared, roughly 2^n, or roughly n^n? Now, none of these four answers is exactly right, but one of the four is a lot closer to the exact expression than the other three and I'm asking you, which of them is it? Alright. So the correct answer is the third one, 2^n. A graph of n vertices has essentially 2^n cut, so there's an exponential number of cuts there's a lot of them. So why is this true? Well, in effect you can imagine making a binary decision for each of the n vertices. They either go into A. What were they going to be? So n binary decisions results in 2^n different outcomes. Now why is this slightly incorrect? Well, in fact, a cut has to have two non-empty sets. A is not allowed to be empty, B is not allowed to be empty, so that rules out two of the possibilities. So actually, strictly speaking, it's 2^n - 1 different cuts of a graph. So what we're going to do next is we're going to state and prove three easy facts about cuts in graphs. Once we have these three easy facts, we will be able to prove the claim at the beginning of this video, namely the Prim's Algorithm always outputs a spanning tree. The first of these three properties about cuts, I'm going to call the empty cuts lemma. The point of the empty cut lemma is to give us a characterization that is a new way of saying when a graph is connected. So in particular, I'm going to phrase in terms of a graph not connected. And the claim is that a graph is not connected if and only if we can find a cut of the graph that has no edges crossing it. So remember how we defined a graph being connected, that means for any two vertices in the graph we can find a path in the graph from one vertex to the other. So what we're saying is that being not connected, that is, there existing a pair of vertices with no path between them is equivalent to there being a cut with no crossing edges. So let's go ahead and prove this real quick. So as an if and only if statement, really this proof, we have to do in two parts. First, we have to prove that assuming the first statement, we can derive the second. Then we have to show that assuming the second statement, we can derive the first. I think the easier direction is to assume the right-hand side and then derive the left-hand side. So let's start with that one. That is, consider a graph G so that there's a cut, A, B with no edges of G crossing this cut. The plan is to exhibit a pair of vertices that do not have a path between them, there, thereby certifying that the graph is not connected. So, it's pretty easy to figure out which pair of vertices we should look at, just take one vertex from each side of the cut which has no crossing edges. So why is it that there's no path from U to V in the graph G? Well the path from U to V would surely have to cross the cuts, A, B, but there's no edges available for crossing the cut. So therefore, this path from U to V cannot exist. So that completes the first part of the proof. We assume the right-hand side, we derive the left-hand side, now we start all over again, but we assume the left-hand side and we have to prove the right-hand side. So by virtue of, by the assumption that the graph is not connected, there has to exist a pair of verticies U and V that have no path between them. We are now responsible for exhibiting some cut A, B such that no edges of the graph G crossing. So where are we going to get these sets capital A and capital B from? Well, here is the trick, which is going to make the proof go really nicely. We define the set of verticies of capital A to be those reachable from U in the graph G. Another way to think about this is that capital A is simply used connected components in the sense that we discussed in part 1 of the course. Now because we want to cut and a cut is our partition, we better well put in the group, capital B, all of the verticies that are not in A. If you like, this is all of the connected components other than the one that contains U. Note that by definition, U is in capital A, certainly U is reachable from itself. And by assumption, V and U are not reachable from each other, so V is going to be in capital B. So neither of these sets is non-empty. This is indeed a bonafide cut of the graph G. All that remains is to notice that there are no crossing edges across this cut. And why is that true? Well, if there was an edge crossing the cut A, B with one endpoint in A, one endpoint in B. Well, by definition, there are paths from U to everything else in A, so if there is any edge sticking out of A, that would give us a path to some vertex in B. But, B definition of vertices not reachable from capital A, so that's a contradiction. So again, the point is that if there were edges crossing this cut, then we can expand A and make it even bigger. So therefore, there aren't any edges crossing the cut. The cut is empty, that's what we needed to prove. Assuming the graph was disconnected, we have exhibited a cut, A, B with no crossing edges. So that wraps up of the first of our three facts, and in fact, the most difficult of our three facts about cuts in graphs. And again,, what did the empty cut lemma say? It gives us a new way of talking about whether or not a graph is connected. So it's disconnected if and only if there's an empty cut. It's connected if and only if there are no empty cuts. So that's the keypoint from this slide. Let's now knock off the other two facts we're going to need. The first one I'm going to call the double crossing lemma. In essence, what the double crossing lemma says, is that, if a cycle in a graph crosses a cut, then it has to cross it twice, it cannot cross it only once. So pictorially, we look at a cut of a graph, so there's the two vertex groups A and B. By hypothesis, there's some edge E with one endpoint in each side, and by assumption, this E, this edge E, participates in some cycle that we're calling capital C. And if you look at the picture, you realize that the claim in this lemma is obvious, that, because the cycle has to loop back on itself, if it has an edge with one endpoint on either side, there has to be a path connecting the two dots, connecting those two endpoints back to each other and that path has to cross back for, over this cut A, B. Indeed, the double crossing lemma is a special case of a stronger statement which is equally easier to see, which is that if you take any cut of a graph and you take any cycle you know, it starts and ends at the same point, then it has to cross this cut an even number of times. It might cross it 0 times, but it's not going to cross it once. It could cross it twice. It could cross it four times, if it crisscrosses back and forth. It could cross it six times, and so on. But if it crosses it strictly more than 0 times, then it has to cross it at least twice. That's the point of the double crossing lemma. So, we'll use this in its own rights later on. But I'm also, for the moment, interested in easy corollary of the double crossing lemma. I will call this the lonely cut corollary. Let me tell you the point of the lonely cut corollary. In general, in these spanning tree algorithms, to ensure that we output a spanning tree, then we have to, in particular, make sure we don't create any cycles. The point of this corollary is it's a tool to argue that we don't create cycles. So how can we be sure that an edge doesn't create cycles? Well, here is a way. Suppose there's a cut, so we're looking at an edge E, suppose we can identify a cut A, B so that edge E is the only cut crossing it, it's the lonely edge crossing this cut. Well then, by the double crossing lemma, there is no way this thing is in any cycle. If it were in a cycle and a cross to cut, that cycle would have to cross it again and it's edge wouldn't be lonely, it would have company. So if you're lonely on a cut, it mean's you cannot be in a cycle. So now we've got all of our ducks lined up in a row and we're ready to prove the first part of the correctness of Prim. That is, we're ready to argue that Prim's algorithm, given a connected graph, outputs a spanning tree. Again, for the moment, we're making no claims about optimality, that will be in the next video. So we're going to make this argument in three steps. And for the first step, you might want to go look again at the pseudocode of Prim's algorithm just to remember what the notation was. The first step, we're just going to notice that the semantics of the algorithm are respected. So the algorithm maintains two different sets throughout its evolution. On the one hand it maintains a set capital x, intended to be the vertices spanned so far. The other hand, it maintains a set of edges, capital T, the edges that have been picked so far. And the intent was that the current edges capital T always spans the current vertex at capital x. So the first thing is just to verify that that is in fact true. This I'm not going to prove formally. In my experience, students find this kind of obvious and the intuition is correct. if you want a rigorous proof, go go ahead and fill in the details yourself. It's a straightforward induction with no nasty surprises. [SOUND] Now, we're trying to argue the output of this algorithm is a spanning tree. So let's recall what that means. What is it that we have to check? So there's two properties. First of all, there can't be any cycles, there can't be any loops. Second of all, it has to span all of the vertices. It has to be a path inside the tree edges from any vertex to any other vertex. So let's go ahead and prove both those things in reverse order. So, the second step of the proof is going to be to argue that the algorithm outputs something which does span all of the vertices. So at the end of the day, we'll have a path from any vertex to any other vertex using only the edges in our chosen set, capital T. Now, by part one of this proof, all we need to prove is that the algorithm halts with capital X equal to capital V, then we know that capital T spans everything in V. So how could that not happen? How could Prim's algorithm somehow halts with this spanned vertices capital X, not being all of capital B,? We'll go back and check out the pseudocode and look at the main wild loop. So every wild loop, every iteration, we add one new vertex to capital X. What could go wrong? The only thing that could go wrong would be is if some iteration, before we're spanning everything, when we scan the frontier around capital X, there aren't any edges. That's the only way we can fail to increase the vertices in capital X in a given duration. But what would that mean? What would it mean if in some iteration we couldn't find edges with one endpoint in capital X and the other endpoint in V - X? Well then we would have exhibited an empty cut. The cut X, V - X would have no crossing edges. And now we can use the empty cut lemma, which says if there's an empty cut, then the graph is disconnected. But by assumption, we're working with a connected input graph, so that can't happen. Okay? So the algorithm never gets stuck, we always increase capital X by one vertex because the original graph was connected, that means that halt was something spanning all of the verticies. For the final step, we need to argue that Prim's algorithm never creates any cycles in the edges that it, it's choosing capital T. So, why are there no cycles? Well, what we're going to do is we're going to talk about each edge in turn, the Prim's algorithm adds, and argue that whenever a new edge gets added, there's no way that edge creates any cycles in the set capital T. And, to see why, take a snapshot of the algorithm of some given iteration, to the sum current set capital T, and there's some set verticies capital X that the edges in T span. V - X to the verticies not yet spanned by T and of course we can think of X, V - X as a cut of the graph. And at this moment in time, at this snapshot, the edges of capital T, they're all of one type. They all have both of their endpoints inside capital X, none of them have any endpoints inside V - X. So in particular, none of the edges chosen thus far cross the cut X, V - X. That's by construction, they only span the verticies of X. Now what type of edge is going to get added in this iteration. Well, Prim's algorithm searches only over edges that have one endpoint inside X and one endpoint outside. That is, it searches only over edges that cross the cut X, V - X. So the edge that gets added in this iteration is going to be a trailblazer for this cut. None of the edges yet shows and cross the cut, but the edge showed in this iteration will definitely, cross the cut. So the moment edge E gets added to the tree capital T, it is going to be lonely across the cut V sorry, X, V - X. So by the lonely cut corollary as the sole member crossing this cut in capital T, it cannot possibly participate in any cycles. Remember, if it participated in a cycle in capital T, that cycle would have to cross this cut somewhere else. But there aren't any other edges crossing this cut, this is the only one. So that's why when we add a new edge, there's no way it can create any cycles. It's the sole member crossing this particular cut.

# 20-4-Correctness Proof II

Alright. So now that we've completed our warm up by showing that at the very least, Prim's algorithm outputs a spanning tree. Let's move on and actually show it outputs a minimum cost spanning tree. And to prove this theorem, we're going to have to tackle head-on the kind of crisis which you always face when designing a greedy algorithm. So in a greedy algorithm, you're making an irrevocable decision, like in Prim's algorithm, we're including an edge in our tree and never revisiting it later. And, how can you be sure that you're not making a mistake? How can you have a guarantee that the decision you're making seemingly myopically right now is actually a good decision won't come back to bite you later? So it turns out for minimum spanning trees, there is a beautiful condition which tells you when you're guaranteed to not regret including an edge into a spanning tree, but guarantees when an edge has to belong to the minimum spanning tree. So that's called the cut property, it's the subject of the next slide. So this is a cool enough property that we're going to bestow it not just with all caps but even with a box. Now, that's a pretty good property. So what does it states? Well, consider an edge of a graph, an edge that we are wondering if it's safe to include it in the tree so far. So here is this sufficient condition guaranteeing that you won't regret including this edge in the tree so far. The condition is stated in terms of the cut. So suppose you can find a cut, A, B with the property that amongst all edges in the graph G, that happened across this cut, the edge E is the cheapest edge crossing this cut. Okay? So not only should edge E cross this cut, A, B, but it should cheapest such edge. If this condition is met, then we definitely want them include the edge E in our solution. Indeed, the edge E has to be a member of any minimum spanning tree of the graph. So in this video, we're going to assume that the cut property is true. It is by no means obvious. It definitely requires a proof. I'll give you the proof in a separate video. It's not, it's a little bit tricky. It's based on a subtle exchange argument. For this video, we're going to assume that it's true, however, and we just want to be quiet, so that we want to figure out what it's good for. Now, I will soon show you that it actually implies correctness of Prim's algorithm. But just to get a feel for it, let's look at a much simpler graph. Let's just look at a four cycle. Four nodes, four edges with edge costs 1, 2, 3, and 4. So let's look at, let's look at a few cuts. So, let's look at the cut. We're on one side of the cut, I put the upper right vertex, and on the other side of the cut, I put the other three vertices. So there are two edges crossing this cut, the edge that has cost 1, the edge that has cost 2. So the edge with cost 1 is the cheapest edge crossing this cut, so by the cut property, the edge of cost 1 has to be in the MST. Okay. So we looked at one cut and both the cut property [INAUDIBLE} to stick in the MST. That was pretty cool. let's look another cut. Let's look at a cut where on one side, we just put the bottom right vertex, and on the other side, we put all Now this cut has two edges crossing it the edges that have cost 2 and cost 3. The edge of cost 2 is the cheapest edge crossing this cut. So by the cut property, it has to be in the MST. So that's cool. So we know the two has got to be there. Now let me point out something interesting that's happened, which is that, it is not the case that this edge of cost 2 is the cheapest crossing, every single cut that it crosses. Remember when we looked at cut number 1, this edge of cost 2 was actually the most expensive edge crossing that cut. But, we found a different cut that is the cheapest crossing and that's enough to justify the cut property. So in other words, all that's important for the cut property, I just got to find you one cut for which an edge is the cheapest, that's enough to conclude its presence in the MST. So similarly, we can look at a third cut just consisting of the bottom left vertext and the other three vertices. And it's the same story, there are two edges crossing this cut, the edge of cost 3, the edge of cost 4. The edge of cost 3 is the cheapest edge crossing this cut, so we know it's got to be in the MST. And again, when we look at cut number 2, it didn't tell us whether or not the edge of cost 3 is in the MST, but when we looked at cut number 3, that was enough to conclude that the edge of cost 3 has to be in the MST. So there we go. So we could use the cut property that construct an entire MST. On the other hand, there's no way to use the cut property to try to justify the edge of cost 4. Any cut that you pick for which the edge of cost 4 crosses, there will be some other cheaper edge crossing it. So you can never use the cut property as one would hope to justify the inclusion of the edge of cost 4 and you'd better not be able to, because 4 is not in the MST. Now a quick side note, some of you might be wondering when I wrote in the conclusion of the cut property, I said the MST of G, so that would seem to indicate that the minimum spanning tree is unique. So that deserves a quick comment. so first of all, if the edge costs are not distinct, if you have ties between edges, then you can certainly have multiple different minimum spanning trees and you have to state the cut property a little bit differently. But again, in the lectures we are just going to assume distinct edge costs, so that's not a problem. And in fact, something that will be a consequence of the next slide, we'll notice that the minimum spanning tree is unique with distinct edge cost. It's not obvious, but we'll prove it shortly. All right. So what I want to do to finish up this video is I want to assume that the cut property is true. And then, from that, I want to derive, I want to argue that Prim's algorithm is correct, always outputs an MST. The proof of the cut property is non-trivial and deserves its own video, which you can see separately. All right. So given the tools that we've developed, this argument is actually going to be quite short. So let's assume that the cut property is a true statement and let's begin by building on the previous video. The previous video argued that Prim's algorithm outputs a spanning tree, didn't argue it was a minimum one, but it argued it's a spanning tree, it spans all the vertices and has no cycles. Let's call the output of Prim's algorithm at the end of algorithm T star. Now, stare at the cut property, stare at the pseudocode of Prim's algorithm. What happens in each iteration of Prim's algorithm? Well, we have our set capital X, that's what we spanned so far. There's the rest of this stuff, V - X, so that's a cut X, V - X. What does Prim choose to include next? Well, in brute-force searches through the edges, the cross is cut and it adds the cheapest one of them. Well, that is right in the wheelhouse of the cut property. What does the cut property says? It says cheapest edges crossing cuts have to be in the MST. So they just fit together beautifully. Prim's algorithm explicitly picks an edge at each iteration which satisfies the hypothesis of the cut property and therefore has to be in the MST. So remember, the conclusion of the cut property says edges so justified must belong to the MST. So if everything in T star is justified by the cut property, then everything in T star is in the MST so T star is a subset of the MST. But T star, of course, as we have argued, is already a spanning tree in it of itself. And, if you add more edges to T star, it's no longer going to be a spanning tree, because you are going to pick up cycles, right? If you ever have something that is connected, there is a path from each pair of vertices, and you add a new edge, you are going to close a path, you're going to get a loop. Okay? So T star is already a spanning tree, and you can't have anything bigger and still be a spanning tree. So therefore, this has to be the minimum spanning tree, there cannot be anything else. So for this reason, T star must in fact be the minimum cost spanning tree of the graph. Since the input graph was arbitrary, assuming only it was connected, this completes, assuming the cut property, the proof of correctness of Prim's minimum spanning tree algortihm.

# 20-5-Proof of Cut Property [Advanced - Optional]

Okay. So to this point we've proven the correctness of Prim's minimum spanning sheet algorithm under an assumption, under an assumption that the cut property is true. So, the purpose of this video was to supply the missing proof to convince you of this cut property. Let me remind you where we stand. So through all the minimum spanning tree videos, we're assuming distinct edge costs. All of this can be extended to edge costs with ties. In particular, there's a version of the cut property when the edges have ties, but we're just going to focus on the main ideas which were exposed already with distinct edge costs. So what does the cut property say? Well, it's meant to be a guarantee that an edge is safe to include on, in your tree so far. So it justifies an iteration of a greedy algorithm like Prim's algorithm. Specifically, consider an edge of a graph, and suppose you can find some cut A, B. So that, amongst all the edges that are crossing this cut, E is the cheapest. So E, the edge E has to not just cross this cut, but it has to be cheaper than any edge that crosses this cut. If you can find just one cut of this form, so that E is the cheapest crossing edge, then it's definitely not a mistake to include E in your tree. You definitely want it. It is in the MST. So this is a non-trivial claim and, let's turn to the proof. At a high level, the plan will be not that different than the correctness of our greedy scheduling algorithm for minimizing the weighted sum of the completion times. That is, we're going to use an exchange argument embedded in a proof by contradiction. [COUGH] The type of contradiction will be of the same form. We'll start with an optimal solution. Suppose it doesn't have the property that we want it to have, and then we do an exchange to make it even better, contradicting the assumption that this solution is optimal. So specifically, if we argue by contradiction we assume, that the cup property is false. So let's just make sure we understand what that means. If the cup property is false, then there's a graph and there's an edge, which actually is the cheapest crossing some cut, and yet, that edge does not belong to the minimum cost spanning tree T star. The plan then is to exchange this missing edge E with some edge that isn't a tree T star, which is more expensive, thereby getting a better spanning tree providing the contradiction. So, this idea currently is a little bit hand-wavy. To really execute it, we have to specify which edge we're going to exchange the edge E with. That's a subtle point and we'll develop it over the next couple of slides. So let's begin with a sort of first cut attempt at an exchange argument. So what's the world look like? Well, we have some cut of a graph. So at one blob, the vertices is A and then the rest of the vertices are in this blob B. this is the cut for which edge E is the cheapest. And by our assumption in this proof by contradiction, this cheapest edge E does not belong to the minimum spanning tree T star. That said, I claim while T star may not have this edge E to cross in this cut, it better possess some other edge crossing this cut A, D. So why is that true? Well, suppose the opposite, suppose in fact T star did not have any edge crossing this cut A, B, well then, T star wouldn't be connected. It wouldn't span all the vertices, right? Remember our proof of empty cut lemma, so if you had this empty cut and there's no way to have a path from one side to the other side. Okay? But that's spanning trees have to have paths between each pair of vertices. So T star as a spanning tree have to contain something from this cut, by assumption it doesn't contain edge E. So it contains some other edge, let's call it F, that crosses this cut. Now of course, since E is the cheapest edge crossing this cut and F is some other edge crossing this cut, F is strictly more expensive than E. And at this point, we seem beautifully set up to execute the desired exchange arguement. We have the edge that the optimal solutions missing. We have a canvid replacement edge F, which is more expensive. So if we swap E and F, hopefully we get a new spanning tree that has strictly smaller cost providing the desired contradiction. But, things are more settled than they were with these scheduling applications. The reason being is that schedules are simply sequences of jobs. So whenever you do an exchange of two jobs, it's clear you get another schedule. But spanning trees and graphs are subtle objects and there's a question, if we take a spanning tree and we add one new edge and delete an edge, do we get another spanning tree of the graph or not? So the following quiz is going to ask you to think about that question carefully. Okay. So what we wish that the answer to this quiz was, was either answer A or answer C. So A would be the cleanest solution. If it were always true that when you take a spanning tree, you take an edge outside of the spanning tree and then you swap those two edges, you get a new spanning tree, then in fact, our proof of the cut property would be done, right? We would just go on that previous slide. We would rip out the edge F from the spanning tree. We'd plug in the edge E, because E costs less than F, we'd get a spanning tree which was cheaper. And we'd be done, that would be our contradiction. Now if A wasn't true, we'd still be okay if C was true. If maybe not every swap yields a new spanning tree, but at least if you're swapping in the edge that's the cheapest crossing some cut, you get a spanning tree. Then we'd also be golden, because in fact, we're only are trying to execute the swap, the exchange, using an edge, which is the cheapest crossing some cut. If you go back to the previous slide, you'll see that was the only case we needed this fact to be true. Unfortunately, the correct answer to this quiz is D. You need not get a new spanning tree when you execute an exchange, even if you're plugging an edge which is the cheapest edge crossing some cut. So to understand this better, let me the picture that we had on the previous slide. We had our cut A, B, we had our cheapest edge E which by assumption does not belong to the spanning three T star, but we observed that T star has to contain at least one other edge crossing this cut because it's connected and we called that F. And we're wondering if swapping E and F yields a new spanning tree or not. So, to reason about this, let me just draw you what the rest of the spanning tree might look like. So in this picture, this spanning tree T star is given by the pink edges. And it's just to this path of five edges on the six vertices. So, what happens if we exchange E and F? Well unfortunately, something bad happens. So we certainly get a new subgraph of five edges, after all, we just subtracted one and added one. But this new spanning, this new subgraph fails to be a spanning tree. It fails on both counts. First of all, it obviously has a cycle and it's a four cycle and secondly, it's not connected. The upper right vertex is just totally disconnected from the rest of the rest of the vertices. So that's no good. That's an exchange which just does not produce a feasible solution and it is therefore not useful in our proof by contradiction. Now, if you just want to salvage some hope from this seemingly promising proof plan, we could take solace from the fact that there is not just one pink edge crossing the cut A, B. F isn't the only one, there's actually this other one on the bottom. so let me call that E prime. Now, E being the cheapest edge crossing this cut overall. Not only is E cheaper than F, it's cheaper than E prime also. So in some sense with our motivation, we could care less which edge we exchange E from crossing this cut, because it's cheaper than all of them. And we see that at least in this example, swapping with E prime yields a good solution, yields a feasible spanning tree. So what have we learned? What we've learned is that if we want to execute this exchange argument, we cannot blithely exchange with any edge of T star that crosses this cut. So the best case scenario, so what we're hoping is true that we can always find some suitable edge, like E prime on the previous slide. So that when we execute this swap, we do in fact, get a spanning tree. And I'm happy to report that we can indeed always do this. So what I need to explain is the procedure by which we exhibit edge, this edge E Prime, which doesn't get us into trouble after we swap, which still gives us a spanning tree after the swap. So let me explain the procedure by which we identify this magical edge E prime that we can swap with and still be a spanning tree. So here's the way to think about it, so we've got this spanning tree T star, we've got this edge which is not yet in T star. Now, if we just plug E into T star, we're going to get a cycle. Why? Well, a spanning tree, remember, it has a path between each pair of vertices. So if this new edge, maybe its endpoints are U and V. T star already has a path between U and V, so when you plug in this new direct edge between U and V, it closes the loop, it gives you a cycle. So let's go ahead and call that cycle capital C. Let me also redraw the picture from the example on the previous slide so you can see what that cycle was in that special case. Now, here's the pattern to notice about this cycle capital C, at least in this example, which is that the lousy edge, the edge F, for which when we swapped, we didn't get a spanning tree, that's off of this capital C. Whereas, the good edge, the edge where we could do a swap and get a spanning tree, E prime that's on this same cycle capital C. And that turns out to be true in general. So, when you add the edge to the spanning tree and you get a new cycle, that cycle is what furnishes the candidates for swaps that will give you a new spanning tree. So the one lingering concern then, we have this cycle. We would, all edges of the cycle are going to be good candidates for the swapping. Wee just need to make sure that there is some edge that actually crosses this cut A, B like the edge E prime does in the picture. But here, we're going to rely on a fact from a previous video, the double crossing lemma. Remember the double crossing lemma says, that if you have a cycle that crosses a cut at least once, then it has to cross it twice. All right. So if it'd cross once, then it has to loop back around, then in looping back around, it's going to cross it a second time. So, in this cycle capital C, we know it crosses the cut A, B once, that's because it includes the original cheapest edge across the cut E. So, it's got to cross it a second time. There's got to be an E prime in the cycle crossing the cut and that's going to allow us to do the swap and get a new, cheaper spanning tree completing the contradiction. So, just to spell things out in a little more detail. So what we do is we first say we use the double crossing lemma. So, we have this reported minimum spanning tree T star. We have this cheap edge E knot in it. We plug E into the spanning tree, we get cycle, we call the cycle capital C. The cycle crosses the cut A, B once, through the edge E. It crosses it a second time by the double crossing lemma. We're going to call that edge E prime. Since E prime crosses the same cut as E, we know that E prime is strictly more expensive than E. Remember we use the cheapest one crossing this cut, A, B. So now what we do is we execute the swap with this new edge E prime. So E prime in T star. The cheapest edge, E, is not in T star so we can swap them. We can take, we can rip E prime out, we can stick E in. Now something I want you to think through carefully at home, convince yourself this is true, is that because we plucked E prime from the cycle, this new tree which I'm going to call capital T, this is a spanning tree necessarily. You know, intuitively, the reason being, you plug in E and you get this one cycle involving E, and then when you rip out E prime, you destroy the cycle. And because it's on a cycle, you don't destroy connectivity between any pair of veriticies, there is still one path between each pair. But make sure you believe that, convince yourself at home. And once you're so convinced, you will also realize that we've finished a proof. We've executed our proof by contradiction. since E was the cheapest edge crossing the cut, and E prime is another edge crossing the cut, E is got to be cheaper. Since T differs from T star only in the swap of these two edges, it's aggregated cost has gone down and that contradicts the purported optimality of T star comp completing the proof of the cut property.

# 20-6-Fast Implementation I

So at this point we understand, Prim's algorithm and we also know why it's correct. That is why it always computes the minimum cost spanning tree of a graph. So in this video, we're going to turn to implementation issues and running time analysis. We'll begin by just analyzing the straightforward implementation of Prim's algorithm. That's already reasonable. It's polynomial running time, but not especially close to linear. Then we'll see how a suitable deployment of heaps very much in the way that we did for Dijkstra's algorithm leads to a blazingly fast, near linear running time. So let's briefly review the pseudocode for Prim's algorithm. Recall that Prim grows a tree one edge at a time spanning one new vertex at each iteration. So it maintains two sets, capital X, a set of vertices that have spanned so far, and capital T, these are the edges we've committed to thus far. They start out by just being some arbitrary vertex, little s, and the empty set, and in each iteration of this main while-loop, we add one new edge to the tree. And whatever new vertex that edge spans, we add that to capital X. The algorithm terminates when we're spanning all of the vertices and as we've seen, it halts not just with a spanning tree but with a minimum cost spanning tree. So suppose we just literally implemented this algorithm as is, what would be the running time? Well, the initialization stick, step takes only constant time, so let's ignore that. So let me just have this one loop. So let's just ask how many iterations does the loop take and how much time is needed to execute each iteration? Well, the number of loop iterations is going to be exactly n - 1, so, where n is the number of vertices. X starts out with one vertex and terminates when it has all n vertices. How much work do we need to implement each iteration? Well, essentially, what each iteration is doing is a brute-force search through the edges. It's looking for the edges that have one endpoint inside X and one endpoint outside, and amongst those, it just remembers the cheapest. And it's easy to see that you could implement each iteration in O of m time, where M is the number of edges. For example, you can just, with each vertex associate a Boolean variable that keeps track of whether or not it's in this capital X, that way when you see an edge, you know whether it's crossing the frontier or not in constant time. So putting it together, O of m iterations with O of m works for each gives us a running time of O of m times n. So this running time is already nothing to sneeze at. As we discussed, graphs can have an exponential number of spanning trees. So, this algorithm is doing far less work than examining each of these spanning trees. It's homing in in polynomial time, to the minimum cost point. So that's pretty cool. But remember the mantra of any algorithm designer worth their salts, confronted with a solution, you should always ask but can we do better? And can we do better than running time O of m times n? We can as we'll see in the rest of this video. The big idea for speeding up Prim's Algorithm is exactly the big idea we used in part 1 to speed up Dijkstra's algorithm, namely we're going to deploy a suitable data structure. So, what data structure seems like it might be a good idea for making Prim run faster? Well, what's happening in the main workhorse while-loop of Prim's algorithm? Over and over again, we keep meaning to do a minimum computation amongst all edges crossing the frontier, we need to find the cheapest one. So, the question we should ask ourselves is what kind of data structure would facilitate, would speed-up repeated minimum computations. And if you recall from part 1, we have a data structure where that's exactly what it's raison d'etre is, the heap, the meaning of life for a heap is to speed-up repeated minimum computations, just like in Prim's algorithm. So let me just remind you briefly, what are the operations exported by heap data structure and what is the running time? So first recall that a heap contains a bunch of objects, and each of those objects should have some key value from some totally ordered set, like a number, like for example, an edge cost. So what can you do with a heap? Well, the salient operations for our purposes today are, first of all, you can insert new stuff into the heap with their, whatever their key value is. You can extract the object with the minimum key value. And you can also delete stuff from the middle of the heap. And all of these can be done in logarithmic time, logarithmic in a number of objects stored by the heap. So it's not going to be important for us today to know how heaps are implemented and what they look like under the hood. We're just going to be clients of them. We're just going to make use of these operations and the fact that they run in logarithmic time. But you know, just for those of you who are curious, and/or want to have your memory jogged. Under the hood, heaps are implemented logically as complete binary tree. They're actually laid out in an array, but you sort of think of them conceptually as being in a complete binary tree. And they, they, they satisfy what's called the heap property. And the heap property is to make sure that you know where the object with the minimum key value is. So the actual definition is, every parent should have a key value which is less than that of its children. So as you go up the tree, the key value can only drop and that means you know where the minimum is got to be. It's got to be at the root of this tree orr the front of the array. So that's great. That's how you locate the minimum so quickly in a heap. Now, what do you do when you want to extract the minimum? So you rip off the root of this tree, and now, you have to rearrange the tree to restore the heap property. So you swap the last leaf up to where the root was, you bubble-down as needed to restore the heap property. how do you insert? You put the new object as the new last leaf and you bubble it up as needed to restore the heap property. To delete from the middle of a heap, you just sort of rip it out and then bubble things up or down as necessary to restore the heap property. Again, that's not supposed to, if you're hearing this for the first time, I know this is too fast, this is just to jog your memory for those of you who already learned this in part 1 of the course. For more details, you can go review the appropriate videos there. So now that I've reminded you about the salient properties of heaps. Let's return to the question of how do we deploy them cleverly to speed-up Prim's algorithm. So our intuition is that because we're doing repeated minimum computations in Prim's algorithm, each time that it's while-looped, compute the cheapest edge cross in your frontier, that's sort of in the wheelhouse of heaps. So how should we use heaps? Well, the first idea, which is a pretty good idea, is to use the heap to store edges, right? Because our minimum computation should result in us choosing an edge, so when we EXTRACT-MIN from a heap, we want it to hand us an edge on a silver platter. So it would seem this would be your first thought, that the heap should store edges and that the key value that you use should just be the edge cost, because you want to find the cheapest edge. So this already a quite good idea using heaps in this manner. We'll already definitely speed-up Prim's algorithm relative to the naive implementation. And in fact. and I'll leave this as an exercise for you to work out. using heaps in this way results in an implementation that has, that runs in time big O of m log n. What I'm going to show you instead is not that implementation, but an even cleverer implementation of Prim using heaps. We're not going to see a benefit in the asymptotic running time. This more sophisticated version will also give us m log n running time, but it would give you better constants and it is the version you would want to implement in practice. [SOUND] So, the one slightly tricky point in this exercise is remembering Prim's algorithm, you don't just want the cheapest edge overall [INAUDIBLE] You want the cheapest edge which crosses the current cut that has one endpoint in each of x and v - x. And, when you use heaps in this way, it might hand you in a silver platter and edge which is cheap, but isn't necessarily crossing the frontier. So, you need some extra checks to ensure that you're always finding the minimum edge and that that edge crosses the frontier between x and v - x. So I'll leave it to you to work out the details of this implementation in the privacy of your own home. What I want to spend our time together on instead is this somewhat more sophisticated, more practical way to use heaps. And for those of you who remember our fast implementation of Dijkstra, this will be very familiar to you. It will be the same kinds of ideas that we used for Dijkstra, and the keypoint is, instead of using the heap to store edges, were going to use it to store vertices. So, in a bit more detail, our plan is going to be to maintain two invariants. The first invariant is going to describe what the heap contains. The second invariant is going to be what the key values of those heap object are. So as I said, we're now going to be starting at vertices in the heap, not edges. Which vertices? Exactly the vertices that we don't yet span. The vertices of v - x. The motivation here is that rather than getting on a silver platter, the edge in which to add next to the tree, we're going to get from a heap on a silver platter, the vertex, that we're next going to add to capital X. So the second invariant tells us what the key values of these vertices in v - x are supposed to be. And we're going to define them to be the cheapest cost of an edge incident of this vertex that crosses the current frontier. So, I think a picture will make this definition clear. So, consider some snapshot of Prim's algorithm at some iteration. We have our vertices X that were already spanning. We have our vertices v - x that were not spanning. And remember, the elements of the heap by invariant 1 are exactly the vertices on the right-hand side, the vertices of v - x. So were trying to find the key value for some vertex in the heap. So some vertex v, which is on the right side, which is not in x. And so, what we do is we look at the edges incident on this vertex v that go back to the left-hand side, so, edges incident to v that are crossing the frontier and there may be of course be many such edges. And the invariant we want to maintain is that the key value for this vertex V is the cheapest of all the incident edges crossing their frontier or in this picture the key should be equal to two. There is the niggling issue of how do you define the key if there are no incident edges at all that are crossing the frontier. So maybe you have a vertex w, which is buried deep inside of v - x, and actually, none of the incident edges go back to the left blob at all. So in that case we just define the key to be plus infinity. So given this high level approach to implementing Prim's algorithm using heaps, we now have a few things to think through. So first of all we have to think about how to initialize the heap so that these invariants are satisfied at the beginning of Prim's algorithm. Second of all, we have to check that if these invariants are satisfied, then we can faithfully simulate each iteration of the while-loop in Prim's algorithm, hopefully very quickly. And then third, we have to think about how do we make sure these invariants are maintained throughout the course of Prim's algorithm, so let's do those in turn. So the first thing is how do we set up the heap at the beginning of Prim's algorithm and a preprocessing step, so that both of these invariants are satisfied. Well, at the beginning, X consists just of this single arbitrary star vertex S. V minus X contains the other n - 1 vertices. The key value of a vertex other than S at the beginning of Prim's algorithm, is just the cheapest edge between that vertex and S if there is one, or plus infinity otherwise. So, the thing to think through and make sure you believe this, is that first of all, with a single scan through the edges, so an O of m time, we can compute the key value for each vertex that needs to go in the heap. And then, we just have to insert those n - 1 vertices into the heap. So that's going to cost us O of m time for an edge scan, and then, m log n for the inserts. In fact, for those of you that really know heaps very well, you might know about a heapify operation which allows you to do a batch of n inserts in O of n time because we can do this even faster in linear time but we're not going to need that in this lectures. And also, I claim that this expression m + n log n is bounded above by the expression m log n, at least an asymptotic notation. To see that, remember two things. First of all we're assuming that the input graph is connected, otherwise there's no spanning trees and it's not interesting to talk about minimum spanning trees. Second of all, in any connected graph, the number of edges m is at least n - 1. So asymptotically, m is always at least as big as n and it can be bigger. So you can always replace an n by an m and get a valid upper bound, so that's what we're doing here. The second issue we need to think through is how do we faithfully simulates each iteration of the while loop in Prim's Algorithm, given that these two invariants halt. So this issue is going to work out beautifully really, by construction. We set up our heap and we set up our definition of keys, so that extracting min from the heap and iteration is a faithful simulation of the brute-force search in the naive implementation of Prim's alogrithm. So specifically, assuming that these two invariants hold when we invoke EXTRACT-MIN from this heap, what it provides to us on a silver platter is the next vertex, not yet in X. The next vertex that we should add to X in this iteration. And moreover, the cheapest edge incident to that vertex crossing the frontier is the one that we should be adding to the set T in this iteration. And the way to think about this fact is to think of us as essentially simulating the brute-force search and the naive implementation using a 2-round knockout tournament. So, in the straightforward implementation of Prim, the way we think of it is we just do a scan through all the edges crossing the frontier and we remember the winner, we remember the smallest cost of them all. Here, with a heap, we're doing it in two steps. So first of all, for each vertex on the right-hand side of the cut, for each vertex in v - x, it locally remembers what is its best candidate so what is the cheapest edge incident on that vertex crossing the frontier. So that's kind of round one, so for an edge to be chosen as the winner, at the very least, it'd better be a local winner. It'd better be the cheapest edge crossing the cut that ends at this particular vertex on the right-hand side of the cut. So that's just in a definition of the key of each vertex and encodes the value of the winner localed in that vertex. And then this EXTRACT-MIN is envoking the second round of this 2, 2-round elimination tournament. It's saying, well, amongst all the proposals from the 1st round, amongst all the crossing edges that are locally minimum given it's endpoint, which of them is the cheapest overall? And that's going to be the cheapest edge crossing this cut, the result of this exact min computation.

# 20-7-Fast Implementation II

So the third and final issue to think through is we need to make sure that we pay the piper, that we keep these N variance maintained. We know that if they're satisfied than we have this great way of finding the best edge in each iteration, we just do an extractment But how do we make sure that these N variance stay maintained throughout the algorithm? So, to get a feel for the issues that arise in maintaining of the invariants, in specific, invariant number two, and also to make sure we're all on the same page with respect to the definition of key value of the vertices in the heap. Let's go through an example. So in this example, I've drawn in the picture a graph that has six vertices. in effect we've already run three iterations of Prim's Algorithm, so four of the six vertices are already in capital X, the remaining two vertices V and W are not yet an X, they're in V minus X. So, for five of the edges, I've given them a cost labeled in blue. For the other edges, it's not relevant for this question what their edge costs are. So you don't have to worry about it. So, the question is the following. So given our semantics of how we define keys for vertices that are not in X, so in this case the vertices V and W. What are their current key values supposed to be? So those are the first two numbers I want you to tell me. What's the current key value of V and W? And then secondly, after we run one more iteration of Prim's algorithm. Then what is the new key value of the vertex W supposed to be? So the correct answer is the fourth one. Let's see why, so first let's remember the semantics of keys. What's the key supposed to be? It's supposed to be amongst, all the edges. That on the one hand, are incidents to the vertex. And on the other hand, are crossing the cuts. It's the cheapest cost of any of those edges. So, for the node V, there's four incident edges with costs one, two, four, and five. The one is not crossing the cut, the two, four, and five are crossing the cut. The cheapest of those is two. So, that's why V's current key value is two. For the node V, the node W, it has two incident edges, a one and a ten. . The one is not crossing the cut. The ten is. It's the only candidate crossing the cut, so its key value is ten. So the third part of the question says, what about when we execute one more iteration of Prim's algorithm? So, what is Prim's algorithm going to do? Well, it's going to move the edge with the smallest key from the right hand side to the left hand side. V has a key value of two, w has a key value of ten, so, V is going to be the one that gets moved from the right hand side to the left hand side. So, once that happens, we now have a new set capital X with a fifth vertex, V is now a member, so the new value of X is everything except for the vertex W Now, the key point is that, as we've changed the set capital X, the frontier has changed. The current cut has changed. So of course, it's a different set of edges that are crossing this new cut. Some have disappeared, and some are newly crossing it. The ones that have disappeared are the two and the four and the five. Anything between the vertex that got moved that was already spanning, going to the left hand side has now been sucked inside of capital X. On the other hand, the edge VW which was previously buried internal to V minus X, with one of it's endpoints being pulled to the left hand side. It is now crossing the cut. So why do we care well the point is W's T value has now changed it use to have only one incident edge crossing the cut the other across ten now with a new cut it has two incident edges both the one and the ten are crossing the cut. The cheapest of those two edges is of course the edges of cost one and that now determines its key value its dropped from ten to one. So the take away from this quiz is that well, on the one hand, having our heap set up to maintain these two invariants is great, because a simple extract min allows us to implement the previous brute force search in Prim's algorithm. On the other hand, the extractions screws things up. So it messes up the semantics of our key values. We may, may need to recompute keys for the vertices. So in this next slide I'm going to show you the piece of pseudocode you'd use to recompute keys in the light of an evolving frontier. Fortunately, restoring in varient number two after an extract min is not so painful the reason being is that the damages done by an extract min are local. More specifically, let's think about what are the edges that might be crossing the cut now that were not previously crossing the cut? Well the only vertex whose membership in these sets has changed is V so they have to be edges that are incident to V. If the other end point was already in X then we don't care this edge has just been sucked into X, we never have to worry about it. But if the other end points, so if this edge is incident to v if the other end point w is not an x. Then with V being pulled over the, the left hand side. Now this edge spans the frontier when previously it did not. So the edges we care about are incident to V with the other N point outside of X. And so our plan is just the obvious one, which is for each dangerous vertex. Each vertex incident to V where the other endpoint W is not an X. We just follow to the other endpoint W, and we just recompute its key, and we just do that for all of the relevant W's. So that recomputation necessary is not difficult, there's basically two cases. So this other end point W now it has one extra candidate edge crossing the cut. Namely the one that's also incident on V. The vertex that just moved. So I did this new edge VW is the cheapest local candidate for W, or it's not. And we just take the smaller of those two options. So that completes the high level description of how you maintain invariants one and two throughout this heap-based implementation of Prim's algorithms. So each iteration, you do an extract min, from the extract min you run the pseudocode to restore invariant number two, and you're good to go for the next iteration. So for those of you who want not just the conceptual understanding of this implementation, but really want to get down to the any degree. You want to dot all the I's and cross all the T's. A subtle point you might want to think through is how it is you implement this deletion from a heap. The issue is, is deletion from a heap is generally from a given position. And so here I'm only talking about deleting a vertex from a heap, that doesn't quite tight check. Really what you want to see is delete the vertex at position I from a heap. So really pulling this off, the natural way to do it is have some additional bookkeeping to remember which vertex is at which position in the heap. So again, for the detail-oriented amongst you that's something to think through, but this is the complete conceptual description of the algorithm. Let's now move on to the final running time analysis. So the first claim is that, the non-trivial work of this algorithm all takes place via heap operations. That is, it suffices to just count the number of heap operations, each of which we know is done in logarithmic time. Okay, so let's count up all of the heap operations. One thing we already talked about, but I'll mention it here again for completeness is we do a bunch of inserts just to initialize the heap in a pre-processing step. So after we initialize, we move on to the main while loop. Remember, there's exactly N minus one iterations of that while loop. And in each one, we extract min exactly once. So these were the easy steps. What you should be concerned about. Are, the, heap operations, the deletions and re-insertions that are triggered by needing to decrease the key avertices that are not in X. Indeed, in a single iteration of Prim's algorithm, in a single move of a vertex inside of capital X, can necessitate a large number of heap operations. So, it's important to think, to count these operations in the right way, namely in a edge-centric manner and the claim is that a single edge of the graph is only going to trigger a single decrease key pair of. Operations a single insertion deletion combo. We can even pinpoint the moment in time at which we're going to have this inser, this deletion and reinsertion. It's going to be when the first of the endpoints, so either V or W, the first iteration at which one of those gets sucked into the left-hand side capital X, that's going to trigger the insert-delete, potentially for the other endpoint. When the second endpoint gets sucked into the left-hand side, you don't care, because the other endpoint has already been taken out of the heap, there's no need to maintain its key. So that means that the number of heap operations is almost twice the number of vertices plus almost twice the number of edges. We're again going to use this fact that the input graph is connected and therefore the number of edges is asymptotically at least the number of vertices. So we can say that the number of heap operations is at most a constant factor times the number of edges, M. As we've discussed every heap operation runs in time logarithmic in the number of objects in the heap so that's going to be Log N in this case so we get an overall running time of O of M times Log N. So this is now a quite impressive running time for the really quite non-trivial minimum cost spanning tree problem. Of course we'd love to do even better. If we could shave off the Log N factor and be linear in the input, that would be even more awesome. But we gotta feel pretty good about this running time. Right? This is only a Log N factor slower than what it takes to read the input. This is the same kind of running time we're getting for sorting. So this actually puts the minimum spanning tree problem into the class of four free primitives. If you have a graph and it fits in the main memory of your computer, this algorithm is so fast. Maybe you don't even know why you care about the minimum spinning shaver graph. Why not do it? It's basically cost-less. That's how fast this algorithm is.

# zwk9-Optional

The following problems are for those of you looking to challenge yourself beyond the required problem sets and programming questions. They are completely optional and will not be graded. While they vary in level, many are pretty challenging, and we strongly encourage you to discuss ideas and approaches with your fellow students on the "Theory Problems" discussion forum.

Consider a connected undirected graph G with not necessarily distinct edge costs. Consider two different minimum-cost spanning trees of G, T and T′. Is there necessarily a sequence of minimum-cost spanning trees T=T0,T1,T2,…,Tr=T′ with the property that each consecutive pair Ti,Ti+1 of MSTs differ by only a single edge swap? Prove the statement or exhibit a counterexample.

Consider the following algorithm. The input is a connected undirected graph with edge costs (distinct, if you prefer). The algorithm proceeds in iterations. If the current graph is a spanning tree, then the algorithm halts. Otherwise, it picks an arbitrary cycle of the current graph and deletes the most expensive edge on the cycle. Is this algorithm guaranteed to compute a minimum-cost spanning tree? Prove it or exhibit a counterexample.

Consider the following algorithm. The input is a connected undirected graph with edge costs (distinct, if you prefer). The algorithm proceeds in phases. Each phase adds some edges to a tree-so-far and reduces the number of vertices in the graph (when there is only 1 vertex left, the MST is just the empty set). In a phase, we identify the cheapest edge ev incident on each vertex v of the current graph. Let F={ev} be the collection of all such edges in the current phase. Obtain a new (smaller) graph by contracting all of the edges in F --- so that each connected component of F becomes a single vertex in the new graph --- discarding any self-loops that result. Let T denote the union of all edges that ever get contracted in a phase of this algorithm. Is T guaranteed to be a minimum-cost spanning tree? Prove it or exhibit a counterexample.

# zwk9-prog

## Prog1

In this programming problem and the next you'll code up the greedy algorithms from lecture for minimizing the weighted sum of completion times..

Download the text file below.

jobs.txt

This file describes a set of jobs with positive and integral weights and lengths. It has the format

[number\_of\_jobs]

[job\_1\_weight] [job\_1\_length]

[job\_2\_weight] [job\_2\_length]

...

For example, the third line of the file is "74 59", indicating that the second job has weight 74 and length 59.

You should NOT assume that edge weights or lengths are distinct.

Your task in this problem is to run the greedy algorithm that schedules jobs in decreasing order of the difference (weight - length). Recall from lecture that this algorithm is not always optimal. IMPORTANT: if two jobs have equal difference (weight - length), you should schedule the job with higher weight first. Beware: if you break ties in a different way, you are likely to get the wrong answer. You should report the sum of weighted completion times of the resulting schedule --- a positive integer --- in the box below.

ADVICE: If you get the wrong answer, try out some small test cases to debug your algorithm (and post your test cases to the discussion forum).

## Prog2

For this problem, use the same data set as in the previous problem.

Your task now is to run the greedy algorithm that schedules jobs (optimally) in decreasing order of the ratio (weight/length). In this algorithm, it does not matter how you break ties. You should report the sum of weighted completion times of the resulting schedule --- a positive integer --- in the box below.

## Prog3

In this programming problem you'll code up Prim's minimum spanning tree algorithm.

Download the text file below.

edges.txt

This file describes an undirected graph with integer edge costs. It has the format

[number\_of\_nodes] [number\_of\_edges]

[one\_node\_of\_edge\_1] [other\_node\_of\_edge\_1] [edge\_1\_cost]

[one\_node\_of\_edge\_2] [other\_node\_of\_edge\_2] [edge\_2\_cost]

...

For example, the third line of the file is "2 3 -8874", indicating that there is an edge connecting vertex #2 and vertex #3 that has cost -8874.

You should NOT assume that edge costs are positive, nor should you assume that they are distinct.

Your task is to run Prim's minimum spanning tree algorithm on this graph. You should report the overall cost of a minimum spanning tree --- an integer, which may or may not be negative --- in the box below.

IMPLEMENTATION NOTES: This graph is small enough that the straightforward O(mn) time implementation of Prim's algorithm should work fine. OPTIONAL: For those of you seeking an additional challenge, try implementing a heap-based version. The simpler approach, which should already give you a healthy speed-up, is to maintain relevant edges in a heap (with keys = edge costs). The superior approach stores the unprocessed vertices in the heap, as described in lecture. Note this requires a heap that supports deletions, and you'll probably need to maintain some kind of mapping between vertices and their positions in the heap.