The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or view additional materials from hundreds of MIT courses, visit mitopencourseware@ocw.mit.edu.

ERIK DEMAINE: All right. Welcome to our second lecture on what to do when you have an NP-hard problem. So two lectures ago we saw how to prove a problem is NP-hard. Last lecture we saw if you want polynomial time but you're willing to put up with a not perfect solution, but you want to get within some factor of the best solution, that's approximation algorithms. Today we're going to do a different thing called fixed parameter algorithms. These are going to run an exponential time in the worst case. But not so bad in a certain sense, which we'll get to.

In general, the theme of these last two lectures and this one is that we'd really like to solve hard problems. We'd like to solve them fast, meaning polynomial time. And we would like correct solutions, also known as exact solutions. OK. We'd love to solve NP-hard problems in polynomial time exactly. But that's not possible unless P equals nP. So pick any two. That's the general idea.

This is a bastardization of a joke which is-- sleep, friends, work-- pick any two. That's the MIT motto. Here in algorithms-- hard, fast, exact-- pick any two. So most of this class is about these two-- polynomial time algorithms give you exact things. That's the class P. Last lecture was about hard problems. We drop exactness. We still want polynomial time. We still want to solve hard problems. So this is approximation algorithms.

And what we're doing today is the other combination. So we want exact, but we're going to sacrifice how fast things are. They're not going to be polynomial time in a strict sense, but it's going to be somewhere in between polynomial and exponential. This is an area called FPT for fixed parameter tractability.

So what's this parameter business? In general, the idea is that we really want an exact solution to an NP-hard problem, which means it has to take exponential time in the worst case. But we want to confine the exponential dependence to something called a parameter. OK. We actually use parameters all the time. For example, on a graph, there's two typical parameters you think about-- the number of vertices and the number of edges. If you're sorting an array, the usual parameter you think about is the size of the array. OK. That's all I mean.

A parameter, in general, is just some kind of size or complexity measure. So in general, a parameter is going to be-- we're going to call it k of x-- should be a non-negative integer-- and x is the input. So you're thinking about some problem, like a problem we'll be looking at today is vertex cover, which we saw in the last lecture.

Vertex cover-- you're given a graph. And based on that graph, we're going to define some function of that graph. So this is the input to the problem, and k is just going to be some non-negative integer, which is a function of that input. Just some measure of how tough your problem is. OK.

And what we would like is a running time that is exponential in k, but polynomial in everything else. Polynomial in the size of the problem, in v and E. OK. So that's the general goal is-- polynomial in the problem size-- which we usually call n-- and exponential in the parameter-- which I'm calling-- just going to call k-- in general, you could consider more parameters, but we're just going to think of two-- the overall size of the problem, and some particular parameter that we look at called k.

So if you can achieve this, which we'll call fixed parameter tractability-- I'll define it formally in a little bit, because there's more than one way you might think of defining it. Some are right. Some are wrong. If you can achieve this, what you get is an exact algorithm for your problem that runs really fast provided k is small. So this is sort of a way of saying, well, you know the problem is NP-hard in general, but as long as this measure k is reasonably small, I'm still able to solve it really fast. So it's a way of characterizing a wide family of-- a big subset of the problem that you can solve.

You know that in general you're going to need exponential time, but this gives you a measure of how hard your input is. May not be the only measure. May not be the best one, in any sense. But if you can define a parameter, and you know that in your practical scenarios that parameter will be small, then you're golden. Then you can actually solve the problem in a reasonable amount of time and get an exact solution. No approximation here. So that's the idea.

So that was a parameter. We're also going to define a parameterized problem. This is just a problem plus a parameter. OK. So we already have some notions of problems. We can take any problem that we've looked at before, like vertex cover. And if we just define some parameter, then we get a parameterized problem when we put these things together.

And usually we would write it as something like, oh, take this problem, and then consider it with respect to this parameter. And in general, for a single problem, there may be several natural parameters that you want to care about. Usually there's actually one obvious parameter. So let's do that for vertex cover.

But in general, we can talk about a problem with respect to different parameters. And some of them may be feasible to solve in this sense. Some maybe not. All right. So I'm going to define k vertex cover. This is almost the same as vertex cover. It just has a k in front. But the k means that it's a parameterized problem instead of just a general problem.

So as in vertex cover, we're given a graph G. And I'm going to think of the decision version of vertex cover. So we're given a non-negative integer k. And we want to know-- is there a vertex cover of size k-- say, less than or equal to k-- is there a vertex cover? Remember a vertex cover is a set of vertices that cover all the edges. And we want the size of S to be less than or equal to k. OK.

So every for every edge, we need to choose one of the two end points. We want the total number of chosen vertices to be, at most, k. And so that's a regular decision problem. But for parameterized problem, we also want to define a parameter function. And that parameter function-- guess what? k. Most obvious thing, given that I wrote the letter k here. That's going to be our parameter. OK.

And most problems, a lot of problems, especially decision versions of optimization problems-- like before, we were minimizing the vertex cover-- this is the decision version where we want to decide whether there's one of size of most k. If you can solve this, of course, you can binary search on k, like you did in your quiz. Hopefully.

So that's all good. And a lot of problems have this some non-negative integer floating around. And that's the, kind of the, obvious choice for the parameter. Doesn't have to be the only one. But today, we're just going to look at vertex cover with this parameterization. In your problem set, you'll look at another problem with another natural parameter. This is usually called the natural parameter. But there's no formal definition of natural. That's just intuition. All right. So that's the set up.

Let's do some algorithms. I guess the first note is that k can actually be small. Nice example is a star graph. So you have the vertices, but what's the smallest vertex cover? 1. Everyone's holding up one finger. You choose this guy-- that in the center, that covers all the edges. So it can be that k is much smaller than v, and our goal here is that we're going to get some polynomial dependence in the size of the graph, but we're going to get an exponential dependence on k.

Now there are many different ways you could think of exponential dependence, but let's start with-- what would be the really obvious brute force solution to vertex cover? OK. I want exact. I'm not going to be clever. What's the obvious algorithm to solve this? Yeah.

AUDIENCE: Try any combination of k vertices, and see if it's a vertex cover.

ERIK DEMAINE: Try any combination of k vertices. See if it's a vertex cover. How many combinations of k vertices are there? And choose k. Good. Let's see. I'm a little out of practice. It's been awhile. Close. Off by one. So try all and choose k. I guess, v choose k, subsets of k vertices.

If I wanted to match this definition exactly, I should try all subsets of less than or equal to k vertices. But, hey, if I choose fewer than k vertices, why not add in a few extras until I get up to k. So it's enough to look at v choose k subsets, because-- subsets of size exactly k-- because that will end up giving the same answer as this question. OK. So for each-- test each of those choices for coverage.

So that just means we loop over-- I guess for every vertex in our set, we mark all of the incident edges as covered. And then we go through all the edges, and see whether every one got marked covered. If not, we reset and try the next subset. OK. This is like not smart dynamic programming. You just guess what the subset is. And see if it covers.

This is how you would prove that this problem is in NP. Right? But now we're actually making it an exponential algorithm. So what's the running time of this algorithm? Yeah.

AUDIENCE: E times v to the k.

ERIK DEMAINE: E times v to the k. Good. Must be. So that's obviously exponential. In a certain sense, the dependence on E and v is in the bottom, which is good. And the k is in the exponent, which makes sense. So this is not surprising. We also don't think of it as good. We've defined this to be bad. OK.

In general, we think of a running time like n to the f of k, where n is sort of the overall problem size here. Here n is basically v plus E. And that's the overall input size for a graph. If we have a running time where the exponent of n depends on k, in a nontrivial way, we think of that as a bad running time.

This is a slow algorithm. It's slow because even when k equals 2-- if you have a large graph-- this is probably not something you want to run. Definitely when k is 10, you're completely hosed. This is a very impractical. And the formal sense in which it is impractical is that the exponent in n depends on k. In general, you cannot say-- so I'd like to-- I mean fixed parameter. The whole point is to think of the parameter as being fixed, like a constant. OK.

Now if the parameter is fixed. If you think of it as at most 100, then, indeed, this will be at most n to the 101 or something. So it is polynomial for any fixed k. The catch is that the exponent of the polynomial depends on k. As you increase k, as you increase your bound on k, the exponent increases. I can't say this is an n squared algorithm for any fixed k. OK.

So exponent depends on k. That's the bad case. So the good case, we're going to define, is that the exponent doesn't depend on k. That may seem like a small change. It is a small change. But it's a big one. It's a small change with a big effect. So I'm going to define, let's say, a parameterized problem is fixed parameter tractable-- which, given how many letters that is, we're going to abbreviate to FPT-- if it can be solved in f of k times polynomial in n. OK.

Because this means that the exponent here doesn't depend on anything. The exponent of n doesn't depend on k. OK. So for this definition-- just to be explicit-- I want the constant here to be independent-- of course, it should be independent of n, and it should be independent of k. This can be any function. It's presumably an exponential function, because if this is an NP-hard problem, something's got to be exponential. This clearly is not exponential. So it's got to be here.

So this is a sense in which we're exponential in k, polynomial in n. But it's much better than this kind of running time. OK. We can think about what-- in the sense in which it is much better once we have an actual algorithm of this type. So let's do-- let's try to solve vertex cover in this kind of time. I claim vertex cover is fixed parameter tractable. There is such an algorithm.

And the algorithm is going to look familiar. Very similar to the 2-approximation algorithm that we had last class for vertex cover. So-- but I'm going to give it a different name, which is bounded-search-tree. OK. This algorithm is also going to feel like dynamic programming. Or we're going to use guessing. In general, exponential algorithms, naturally is guessing. But here, when I guess, I have to try all the possibilities.

Here this was one way of trying all the possibilities. We're going to be a little bit more sophisticated in how we try all the possibilities that actually exploits the properties of vertex cover. First line is just like the 2-approximation algorithm. Look at any edge in the graph. OK. Here it is. From u to v. What do I know about that picture? Yeah.

AUDIENCE: One of those vertices has to be in the cover.

ERIK DEMAINE: One of those vertices has to be in the cover. Either u or v or both are in S for that edge to be covered. Now for the 2-approximation, we just put those both in. Here we can't afford to do that because we want an exact solution. So we'll try both options. We don't know which one belongs. Let's guess.

So we know either u is in S or v is in S. Don't know which. So guess. Sorry-- I should, to be clear, mention or both. So we're going to guess, which means we need to try both options. We're going to try putting u in, and then we're going to try putting v in. So let's just see what happens when we try that.

So in the first guess, we say, let's put u in S. OK. Well, if we put u in S, that means we cover all of the edges incident to u. So I'd like to use recursion. I'd like to simplify my problem. Get another vertex cover instance. So in order to do that, I'm just going to delete u and all of its incident edges. We do the similar thing in the approximation algorithm but for u and v simultaneously. So delete u as incident edges. Now we have a vertex cover instance.

There's one other thing. There's a new graph we have. But we also need to update k. Because we just used one of those-- we just added something to S, and then we deleted that from the graph. Which means, in our new graph, effectively k has gone down by 1. OK. So I'll say decrement k. Now I have a new instance. I have a new graph and a different value of k. Recurse this algorithm.

I would say-- I'll call the new graph G prime and the integer k prime. k prime equals k minus 1. And then the second case is do the same thing for v. I won't write the code, exactly the same, but I delete v and it's incident edges. I still decrement k by 1. And I recurse. And then I just return the or of these two answers. So if this one finds a solution, great. I found a solution to the overall problem. This one finds a solution, great. Maybe both return yes. Doesn't matter. In general, I just take the inclusive or of those two Boolean values. That gives me an overall yes no answer to k vertex cover. Cool?

So next question is what the running time is. But you can think of this as a dynamic program. It's just, here we recurse, and we don't bother memoizing. Because, in general, memoization will never help us here. And you may have even thought of algorithms like this in the dynamic programming world. And we just say, well, that's not good enough, because in dynamic programming we want polynomial time. This is like a dynamic program, but the running time is exponential. But it turns out it will be fixed parameter tractable. That's the good news.

Let's think about the running time. So if I draw-- let's draw a recursion tree. Right? This is a divide-and-conquer algorithm in a very weak sense. We start up here with a problem of size n and a parameter k. And we make two recursive calls. OK. We deleted a vertex and maybe some edges. So let's say, we have a new problem of size something like n minus 1. But what really saves us is that k went down by 1.

And we have two recursive calls. Each of them k is 1 smaller. OK. And then each of those has two recursive calls. I don't really know what happens to n. It probably doesn't get that much smaller, but k goes down by another 1. OK. So I'm writing here the size of the problems and the parameters of the problems. n minus 2. k minus 2. OK.

How much time do I spend in each of these nodes? How much work am I doing-- non-recursive work am I doing in this algorithm? Yeah.

AUDIENCE: o of E, right? [INAUDIBLE].

ERIK DEMAINE: o of E. Yeah. Certainly at most order E. Probably at most order v, because there's only at most v incident edges to each vertex. Yeah? Linear time. Doesn't really matter how careful we are here, but I will say-- each of these nodes-- we spend, at most, let's say, order v time. OK.

It happened that v went down by 1. As you can see at these levels. But certainly an upper bound is the original v. In each of these nodes, we spend at most the original v. When does this recursion stop? I didn't write a base case. Help me out. What's a good base case for this algorithm? Yeah.

AUDIENCE: When k equals 0, check if there are any edges.

ERIK DEMAINE: When k equals 0, check if there any edges. When k equals 0, I can't put anything into my vertex cover. So if there any edges, they're not going to be covered. That's bad news. OK. So over here we have base case, k equals 0, check-- or let's say, return whether size of E is not

0. If it's not 0-- sorry-- whether it equals 0-- get it right-- if it equals 0, then the answer is yes. There's a vertex cover. I can cover all of those 0 edges using 0 vertices. That's good.

But when E does not equal 0, there's no way I can cover that non-zero number of edges using 0 vertices in my vertex cover. OK. So that's the base case, which means this recursion keeps going until we get down to k equals 0. We start at k. We end up with 0. So the number of levels here is k. OK. The height of this tree-- this recursion tree is k. So how many nodes are there in this tree? 2 to the k.

So total running time is v times 2 to the k. I guess I should write 2 to the k times v. Hey, that is exactly what I wanted. I got a function of k-- namely 2 to the k. Exponential-- that makes sense. And I got a polynomial in n. Here it's n. The exponent is 1. v is at most n. n us v plus E. Wow. Big improvement. This seems equally simple of an algorithm as this one, but actually it runs a lot faster. OK.

Let me give you a feeling-- I mean this is what we would call a linear time algorithm for fixed k. The exponent here doesn't depend on k. If k is 10, it's a linear time algorithm. If k is 100, it's a linear time algorithm. If k is 100, that might be a little bit beyond what we can run. But you know, k equals 32, 40 maybe, that would probably be reasonable running time, in practice. OK. That's a lot better than before where like k equals 2 or 3. This is probably unreasonable. v is like a billion, say, big graph.

Also from a theoretical perspective, this works even up to k equals log n. If k equals log n, this'll be n squared. That's nice. k equals 2 log n, it's n cubed. OK. So it grows. But we can handle k equals order log n. And this will still be polynomial. In general, with fixed parameter algorithms, it's not always going to be up to log n, it's going to be up to whatever the inverse of this f of k is. That's where we can still be polynomial.

So that's nice. I consider this a good running time. Good in the sense that it follows that definition of fixed parameter tractable. So bounded-search-tree algorithm is good. Brute force algorithm is bad. In this case. Bounded-search-tree is a general technique. You can use it for lots of problems. We're going to see another technique today called kernelization.

But-- Let's see-- before I get there, I want to question this definition. So this definition is nice. It's natural in the sense that it gives you-- it distinguishes between the exponent of n depending on k and not depending on k, which is a natural thing to do. But there's another natural definition of fixed parameter tractability. So let's-- vertex cover-- I think you remember the problem by now.

So let's see-- we have this definition, which is f of k times polynomial n. But I would say that the first time I saw fixed parameter tractability, I thought, well, why do you define it that way? I mean, maybe it would be better to do f of k plus polynomial n. That would be better, right? That would be faster, seems like. So I mean, this is nice in that we achieved this bound, but could we hope for this even better bound. OK?

It turns out these notions are identical. This is weird. The first time you see it. So theorem-- you can solve a problem in this kind of time, if and only if you can solve the problem in this kind of time. So of course, f is going to change. And why don't I label these constants. So we have c up here and some c prime up here. But you can solve a problem in this multiplicative time, if and only if you can solve it in an additive time with a different function and a different constant.

This is actually really easy to prove. The longer you think about it, the more obvious it will be. If you have an instance of size n with parameter k, there are two cases. Either n is less than or equal to f of k, or n is greater than or equal to f of k. Right? It's got to be one of those, maybe both. If n is less than or equal to f of k that means that this running time, f of k times n to the c-- let's see-- n is at most f of k. So this is at most f of k to the c plus 1 power. Right?

I multiply f of k to the c times f of k. When n is greater than f of k, then I know that this running time, f of k times n to the c, well, now I know an upper bound of f of k. I know this thing is at most n. And so this is at most n to the c plus 1. OK. So really I have two scenarios, either I'm bounded by some purely function of k, or I'm bounded by some purely polynomial of n. Which means, in both cases, the running time f of k times n to the c is bounded above by the max of those two things, max of f of k to the c plus 1, n to the c plus 1. OK. And the max is always, at most, the sum.

I'm assuming everything here is non-negative. So I take f of k, c plus 1, plus n to the c plus 1. Boom. That is an additive function of k plus polynomial in n. OK. Rather trivial. This a funny area where you think, ah, this is deep question. Are these things the same? And ends up, yeah, they're the same for obvious reasons.

So for example, we have this linear, basically n times 2 to the k algorithm. If you apply this argument, you get this is, at most, so this n times to the k bound, is, at most, n squared plus 4 to the k. OK. I'm basically just squaring both of the terms. OK. Probably you prefer this time bound, but if you really like an additive time bound, the exact same algorithm satisfies this. OK.

So not that exciting. And in practice, n squared-- it looks like a bad thing, so you'd probably prefer this kind of running time. But there is a sense-- there's a quadratics thing going on here in that we have an n and then we have a function of k multiplied together-- OK-- whatever. All right. So this justifies the definition. This is kind of robust to whether I put a dot here or plus, so clearly this is the right definition. We're going to use dot. You could also use plus, but-- all right.

But there's another thing called kernelization, which, in an intuitive sense, matches this idea of plus. And it also matches an idea that's common practice called pre-processing. If I have a giant graph, and I'm given some number k, and I want to find a vertex cover, well, maybe the first thing I could do is simplify my graph. Maybe there's some parts that are really easy to solve. I should throw those away first. And that will make my problem smaller. So if I'm going to have an exponential running time, presumably, I want to first make the problem as small as I can. Then deal with one of these algorithms. OK. So we're going to do that.

First, I'm going to tell you about it generically. And then we'll do it for vertex cover. So first, let me give you a definition of what we'd like out of this pre-processing procedure. It's going to be called a kernelization procedure. Kernelization algorithm is a polynomial time algorithm. Head back to polynomial time land.

You can think of it as a reduction, but with NP-hardness, we reduced from one problem a to another problem b. Here we're going to reduce from the problem a to the same problem a. It's a self reduction, if you will. But the input to the problem is going to get smaller. So we're going to convert an input. So this is for a parameterized problem. So an input consists of some regular input x and a parameter k. And we want to convert it into an equivalent small input x prime k prime to the same problem. OK.

The problem is fixed, say vertex cover. So we're given an arbitrary input. This would be a graph and a number k. And we want to convert it into an equivalent small input, which is another graph G prime, and another parameter k prime. So equivalent means that the answer is going to be the same. OK. And I want the answer to the problem-- let's say, answer of x comma k to be equal to the answer to x prime and k prime. Again, same problem, but different input.

I'm trying to be a little generic here. It could be-- we're going to think here about decision problems, but this makes sense even for non-decision problems. Whatever the answer is here, it should be the same as the answer is here, because I want an exact solution. I want to solve exactly the problem. I want to compute this answer exactly correctly. So if I can reduce it to some x prime k prime with the same answer, well, now I can just solve x prime k prime. So that's good.

Now what does small mean? We need to define both of these. Small means that the size of x prime, which you might call n prime, should be, at most, some function of k. Cool. So this is interesting. So we started with probably a giant problem, x excise n, and we have a parameter k, which we presume is relatively small. And we convert it into a new input x prime that's very small. Its size is a function of k. No more dependence on n. n has disappeared from the problem. OK.

We start with something a size n. We produced something the size as function of k. And then-- OK-- there's some other parameter k prime-- doesn't matter much what it is. It's going to also be a function of k. So we started with something of size n. We produced something of size k. In polynomial time. Wow. This would be big if we could do it. Because we start with giant problem small k, and we kernelize it down-- so here's the picture with this big thing.

And the intuition is that the hardness of the problem is just from this thing of size k. But there's no thing of size k. Or at least we haven't found it yet, right? The k is the size of the vertex cover we're looking for. But we don't know where that is. It's hiding somewhere in this instance, in this amorphous blob. So it's k. But somehow, magically, this kernelization procedure produces a new problem that's only a little bit bigger than k. OK. Some function of k.

So we take the big problem, we make it down to this small thing. What do you do now? You can run any algorithm you want, any finite algorithm applied to this instance will run in some function of k time. Doesn't matter as long as this is a correct algorithm. If your problem is in NP, there is an exponential time algorithm. You just try all the guesses. So we could use-- we have two of them here. We could run either of these after we've kernelized.

And we would get an FPT time. And, indeed, the FPT would mimic this kind of running time. We do a polynomial amount of pre-processing. That's the kernelization procedure. That's the only dependence on n. After we've done that, the new problem is entirely a function of k. And then you apply any algorithm to that, you'll get f of k running time. Now if we want a good f of k, we should use the best algorithm we have. But, in general, you could use anything. All right. So far, so good.

So we had this theorem that product is the same as plus. In fact, kernelization is the same thing. So these are all the same thing. I guess this one is equivalent to being FPT. that's the definition of FPT. So I'll just write it here. The problem is FPT, if and only if it has a kernelization. This is crazy. I keep introducing stronger and stronger notions of good. And they all turn out to be the same. That, again, gives you a sense of robustness of this definition. And why this is a natural thing to study.

So this sounds crazy. How could I put all of the easy work at the beginning in this polynomial time algorithm and then in the end produce something that is a reasonable size? Again, this proof is going to be trivial. I think everything in this field is either really hard or trivial. I guess that makes sense. So let's first-- so I'm just looking at this inequality-- this implication-- we already did the other one.

The easy direction, of course, is this way. I didn't even do it in this case. If I have an additive running time, it certainly, at most, the product, assuming both those numbers are at least 1. OK. And again, if I have a kernelization, as I said, I could run the kernelization algorithm in polynomial time, and then run any finite algorithm to solve the problem, and I would get some f of k running time. So this is easy.

Kernelize and then run any algorithm on the kernel. Kernel is the produced output x prime k prime. OK. Let's do the other direction. That's the interesting part. So suppose I have it an algorithm that runs, let's say, in this running time. I claim that I can turn it into a kernel. And the proof is going to look just like before. OK. So there are two cases. One is that f of k, let's say, is less than or equal to n.

Actually I want to do the other case first. I think it's a little more natural. Well, it doesn't really matter. They're both easy but for different reasons. the then parts are going to look different. So the first case, if n is at most f of k, what do I do in that situation, in other words, kernelize of this thing of size n, I want to kernelize into something of size f of k? Nothing. I'm done already. So this is the already kernelized case. That's great.

So the other cases, and it's big. That's the more interesting case, of course. n is greater than or equal to f of k. What happens here? Well, just like last time, that means that this running time, f of k is now at most n, that means this running times is at most n to the c plus one. Right? So that means the FPT algorithm that I'm given, because we're assuming here we're given an FPT algorithm, we want to produce a kernel-- in that case, that algorithm runs in n to the c plus 1 time. Which means I can actually run it. That's polynomial time. OK.

So over here I needed a polynomial time kernelization algorithm. If it happens that f of k is at most n, then I can actually run the FPT algorithm, and that would be a valid kernelization procedure. Now the FPT algorithm actually solves the problem. Let's say, it says yes or no, whether the answer to my original question. The kernelization procedure has to output an input to the problem. So what I need to add one thing here which is just-- output a canonical yes or no input accordingly. OK.

If the FPT algorithm-- here I'm thinking about decision problems-- if the FPT algorithm says yes, I'm going to output one instance, one input of the problem where the output is yes. I know that one exists, because this algorithm said yes. So in a constant amount of space, I'm able to write a yes input. Or in a constant amount space, I write a no input. So the new kernel will have constant size, which is smaller than f of k. OK. That's it.

So either I output the same input that I was given. Or I output something of constant size that encodes a yes or no. Kind of trivial again. The catch is that size of the kernel, the size of the output, in general, here is going to be exponential in k, because f of k presumably is exponential in k. That's annoying. This is what you might call an exponential size kernel.

So an interesting question. And exponential size kernels are equivalent to FPT. Something that may not be equivalent is a polynomial size kernel. It would be nice if I start with something that's polynomial in n, and I reduce it to something that's polynomial in k. And then I run something that's exponential on that. But it will only be singly exponential in k, hopefully.

Whereas, if I use this kernelization procedure, if I apply to vertex cover with one of these algorithms, I'm going to get a new thing that's size is exponential in k. If I run one of these brute force algorithms, I'm going to get something that's like doubly exponential in k. That's not so hot, because I know how to do exponential in k. All right. But this is the general idea of kernelization, and why it's not surprising that you can do it.

I have one catch here which is-- this is an algorithm. It compares n to f of k. In order to do this, you have to know what k is. Minor technicality. If you don't know what k is, you can basically run this algorithm with a timer, a stopwatch, and if it's running time exceeds n to the c plus 1, then you know you're not in this case. If it finishes within that time bound, great. You found the answer. If it doesn't finish, then you know you must be in this case, and then you just quit, and output your original input and say, I'm kernelized. Done. Easy. OK. That's a technicality. All right.

So much for general theory. Let's go back to algorithms. Yeah, all this work I want to write down over here. We have a v times 2 to the k algorithm. On the one side. And here, we have an E as v to the k algorithm. Just want to keep a running-- we're going to get a faster algorithm than both of those through kernelization. So I claim that we can find a polynomial kernel-- polynomial-sized kernel-- it's going to be quadratic for vertex cover.

These are hard to find. And there's a whole research industry for finding polynomial kernels. So I'm going to give you some methods. But they're specific to vertex cover. So here's the first thing. This is hard to draw. So here I have a vertex u, and suppose I have an edge connected from u to u. This is called a loop. What can I do from a vertex cover perspective? What can I conclude about this picture? Yeah.

AUDIENCE: [INAUDIBLE].

ERIK DEMAINE: Right. You must be in the vertex cover, because this edge really only has one endpoint. OK. So far, so easy. So what I can do in this case is say, OK, u is in the vertex cover, and then delete u at it's incident edges. So-- and then we have to decrement k. OK. Cool. Seems-- feels familiar.

But in this case, there's no guessing. We just know you must be in the cover. All right. Here's another case. Suppose I have u and v and there are many edges connecting them. This is called a multi-edge. So maybe you just assume your graph is simple. But if you don't assume your graph is simple, we might have these kinds of situations. What can I do in this case? What can I guarantee? Yeah.

AUDIENCE: You can remove all but one edge.

ERIK DEMAINE: You can remove all but one of the edges. Let's just delete all but one. If I cover one of them, I cover them all. Easy peasy. See if you get the other rules. So delete all by 1. In general, we're going to have a bunch of these kinds of simplification rules are guaranteed correct. They don't change the output. But magically, we're going to end up with something the size of function of k. We haven't done much yet. But now we know that the graph is simple, meaning it has no loops and it has no multi-edges. Cool. All right.

Next thing I want to think about is a vertex of degree greater than k. k is the current value of k. So suppose I have a high degree vertex, more than k, edges out going from it. What can I say then? Yeah.

AUDIENCE: You must pick it.

ERIK DEMAINE: You must put it in the cover. Why?

AUDIENCE: Because then you need to cover all the remaining error vertices.

ERIK DEMAINE: Right. Proof by contradiction. If I don't put it in the cover, that means all of these guys are in the cover. There's more than k of them. And the whole goal was to find a vertex cover of size at most k. So you better not put all of these in your vertex cover, because that's more than k. So, therefore, this one has to be in there. This is a cool argument. Simple, but cool. OK.

So any vertex of degree greater than k must be in the vertex cover, which I was calling S. OK. So delete that vertex and its incident edges, decrement k, because we just used something. OK. So just keep doing this. Every time I see a vertex of degree more than k, delete it, decrement k. Now I have a new graph and a new value of k. Look for any vertices whose degree is more than k. If I find one, delete it, repeat, repeat. Keep repeating until you can't anymore.

How much time is this going to take? I don't know. Most quadratic, right? I look at all the vertices. Look at their degrees. I could look over all the edges, increment the degrees. In linear time, I can find whether there's any vertex of degree more than k. Then delete it in linear time. Then try again. And this will happen to most linear time many times because I can only delete a vertex once. So I delete at most v vertices. So overall running time here is like at most v times E, polynomial.

Probably if you're clever, use a data structure to update degrees. You could do this in order v time. But let's not be clever yet. All right. So now, after I've done all of these reductions, I have a graph where every vertex has degree at most k. So it's like a bounded degree graph.

Why do I care about a bounded degree graph? Remember, I drew this example, which was a star graph, where n was large, but k was very small. n was n, n was v, and k was 1. Now a star graph is special because it has a very high degree vertex. In general, if I have a vertex of some degree, say k, and I put it in the vertex cover, it covers k edges. OK. So each vertex in S covers at most k edges, wherever the degree is. So that means you don't get much bang for your buck anymore.

We've already taken care of all the high degree vertices where you get a lot of reward for putting one vertex in the cover. Now this is the new value of k. It may have decremented from before. Every vertex that we could possibly put in the set will only cover k edges. Now we know that we only get to put k more vertices into the set. We know-- we're supposing that sides of S is at most k. So that means that the number of edges, size of E, must be at most k squared. All right. Because every one I put into S covers at most k edges. All of them have to be covered. And so k times k is k squared. Hah, interesting.

That means my graph is small. Now slight catch. There might be a whole bunch of vertices that have no edges incident to them. So I need to delete isolated vertices. Let's say, degree 0 vertices. OK. Degree 0 vertices-- you really don't want to put those into your vertex cover. No point. They don't cover any edges. So delete those.

And now, I still may not have a connected graph, but, in the worst case, I have a matching. I know the total number of edges is at most case squared. That means the total number of vertices is at most twice that, 2k squared. So after all of these operations, I assumed that S was size at most k. And then if I do all of these operations, I get a graph with at most 2k squared vertices, and at most case k squared edges. So the total size of the graph, which I'm calling n, n which is size of v plus size of E is order k squared. 3k squared.

And I assumed that there was a vertex cover size at most k throughout this. So what I do is I run this kernelization algorithm. And I see-- is the graph that I produced size at most 3k squared? If it is, output it. That's a kernelized thing because it's small. If it isn't, if the graph I've produced is still too big, that must mean that this assumption was wrong, which means the answer to the vertex cover problem is no, there is no vertex cover of size at most k.

And so then I just output a canonical no instance, like-- so I mean, this is sort of outside-- but if the newly produced graph-- I'll call it v prime plus E prime is greater than 3 times k squared, then output-- and here I can actually give you one-- let's say, I'm going to output the graph which is a single edge with two vertices and k equals 0. The answer to vertex cover in this instance is no. So this is an example of a constant size, no representative.

So either I get something that's small and I output that, or it's big, in which case I output this thing, which is to say, nope, can't be done. That's kernelization. So I've produced a quadratic size graph, quadratic in k in polynomial time. Question? No. Wow. So this is kernelization at its finest. A polynomial kernel. Polynomial time. We get that down to something the size of polynomial in k.

This is how you should-- if you want to solve vertex cover, you might as well do these reductions first, because they will simplify your thing. And now, if you happen to know your vertex cover is small, then the graph will be small. So now we could run either of these algorithms. OK. Presumably, we should run the better one. But for fun, let's analyze both of them. OK. So I'm going to leave the running times here.

We're going to get a faster vertex cover algorithm from a fixed parameter tractability perspective. So here's a new FTP algorithm. Two of them. First we kernelize. OK. We spent-- I guess-- order vE time. Again, I think you can get that down to order v without too much effort. Obvious. It's not totally obvious. It's a good exercise. Be a good problem set problem. It's not on the problem set. Don't worry. It could be a good final exam problem. Probably a little long. All right.

So now we could-- let's say, option one-- let's use the brute force algorithm after that. The running time of that is E times v to the k. But now E is k squared. And v is also order k squared. Let's not worry about-- actually I do have to worry about constants here, because it's in the base of an exponent. So I do. So we're going to get k squared for the E term. And then v term is going to be 2 times k squared. And that's going to be raised to the k-th power. OK.

So I'll simplify a little bit. This is like 2 to the k times-- I guess-- k to the 2k. OK. It's k to the k squared. Not bad. Overall running time is vE plus this. It's a function of k. It's exponential. Good. We have a better algorithm. We have this v times 2 to the k running time, so we might as well use that one. But the point is-- once you kernelize, you can use pretty stupid algorithms, and you still get really good running times. OK. We'll get a slightly better running time using the bounded-tree-search.

So if we use bounded-tree-search, we have v. v is 2k squared. So here the constant doesn't matter, because there's no exponent. And then we have times 2 to the k. So we're going to get k squared times 2 to the k algorithms. Kind of funny symmetry here. 2 and k are switching roles. Of course, the 2 to the k is the big term. But now it's only singularly exponential in k.

This thing is like 2 to the k log k. This thing is only 2 to the k. So it's better. And this is like k factorial. And this is just 2 to the k. So it's a big improvement. This will be a much more practical algorithm. So we run the kernelization, then we run the bounded-tree-search algorithm. And so the total running time is vE plus k squared 2 to the k.

The story doesn't end here. There are dozens of papers about how to solve vertex cover from fixed parameter tractability perspective. The best one so far-- I'm not going to cover-- but it is based on kernelization. Just more rules. And you get k v plus 1.274 to the k. And some cover's better than 2, but very similar. That's vertex cover. If you have a vertex cover instance, and you know that it's going to have a relatively small vertex cover, these are the algorithms you should use.

Any questions? This ends our vertex cover story. But the last thing I want to do is connect up these two areas. Last class we talked about approximation algorithms. This class we talked about fixed parameter algorithms. They're actually closely related. And so, for example, we will get a fixed parameter algorithm to subset sum, using what we already had last lecture. So, so far today, I've basically been talking about decision problems. But let's think a little bit about optimization problems.

So take your favorite optimization problem. Like any of the ones from last lecture. And let's assume that the optimal solution value-- the thing we're trying to optimize, minimize, or maximize-- is an integer. Assume that OPT is an integer. OK. Now let's look at the decision problem. Whenever you have an optimization problem, you can convert it into a decision problem. You can convert it into a few.

For example, OPT less than or equal to k, or OPT greater than or equal to k. They're all going to turn out to work the same. OPT equal k would also work. Now that's a decision problem, but what I want is a parameterized decision problem. What should my parameter be? k. All right. That's the obvious parameter.

In some sense, we're parameterizing by OPT, but we're adding a layer of indirection. We're saying, well, OPT, but we want to decide whether OPT is less than or equal to k. And let's parameterize by k. That's similar flavor to what we had with vertex cover. If we started with minimum vertex cover, and converted it. Cool. Here's the theorem. This is not going to be as strong as the other things we've seen. No equivalence here.

So it's a one way implication. And I haven't defined this term yet, but it's similar to one we saw last class. If the optimization problem that we started with has an efficient PTAS, an efficient Polynomial Time Approximation Scheme, then the decision problem-- you get from here-- is fixed parameter tractable with respect to k. OK.

So what does EPTAS mean? It's going to look familiar. We're going to take an arbitrary function of 1 over epsilon times a fixed polynomial in n. So last time we talked about PTAS we could have-- you could have something like n to the f of 1 over epsilon. I'm going to consider that bad, as you might imagine, from a fixed parameter tractability perspective. Better would be some function, possibly exponential, if 1 over epsilon times polynomial in n. This is going to be good from a fixed parameter perspective. Although it's about approximation algorithms, not about exact algorithms.

Of course, even better is the FPTASs we saw last time, which is polynomial 1 over epsilon times polynomial in n. That's ideal. If you have an FPTAS, it is also an EPTAS. You just remove-- or you just add one more stroke to the first letter. And you got an EPTAS. And last class we actually saw an EPTAS. For subset sum, we saw 2 to the 1 over epsilon times n. Now, in fact, for that problem, there's an FPTAS. Even better.

But you can see from last lecture why it's nice to have an exponential dependence on 1 over epsilon. And what this is saying is you can do that as long as that exponential dependence is separated from the n part. If it's multiplicatively or additively separated, as you might imagine, it's the same thing, from n, then we call this an efficient PTAS. Not fully, not quite as good as an FPTAS, but close. And as long as you have such a thing, you can convert it into an FPT algorithm for the decision problem.

The way this is typically used-- so this tells us we get an FPT algorithm for subset sum. In fact, because there's an FPTAS, we get a pseudo polynomial time algorithm, which is in some sense better. Anyway. The way this theorem is usually used is in the contrapositive. What this tells us is that if we can find a problem that is not FPT-- and there's a whole theory like NP completeness for showing the problems are almost certainly not fixed parameter tractable-- then we know that there is not an EPTAS.

And this is the state of the art for proving that these kinds of algorithms do not exist. Typically, you look at it from a fixed parameter perspective, and show that probably doesn't exist. Then you get that this probably doesn't exist. OK. Let's prove this theorem. It's, again, really easy. But a nice connection between these two worlds. All right.

So there are two cases-- the optimization problem we're thinking about could be a minimization or maximization problem. Let's say it's maximization, just to be concrete. It won't make too much difference, but it will make a tiny difference in order of-- or the inequality directions. OK. So what we're going to do. So we're given an EPTAS, and we want to solve FPT. We want an FPT algorithm. So what do we do? Well, an algorithm is going to be to run that EPTAS. That's sort of the only thing we can do.

Now the EPTAS-- this is an approximation scheme. It has an extra input which is epsilon. We need to choose epsilon, because we're not-- we're trying to solve it exactly. But there's no epsilon in that problem, so we got to make one up. Let's run the EPTAS with-- anyone have good intuition? What should epsilon be? Yeah. Remind you of this. Tricky. We want epsilon to be small. Yeah.

AUDIENCE: It should be 1 over k.

ERIK DEMAINE: 1 over k is almost right. Anything less than that would work. So I'll use 1 over 2k, but 1 over k plus 1 would also work. Or anything a little bit-- 1 over k-- yeah-- 1 over k plus .00001 something. Anything a little bit less than 1 over k will turn out to work. So, why?

So first of all, how much time does this take? Well, we were given this running time. 1 over this is 2k. So this is going to take f of 2k time times polynomial in n. OK. We need to connect E and k, because we're given-- sorry-- epsilon and k, because we're given something whose running time depends on epsilon not k. Now we're setting epsilon in terms of k, so now the running time is a function of k, not epsilon. And then times n. So this is good. This looks like an FPT running time.

I claim we found that the answer. OK. This is maybe the surprising part. You had good intuition here. And the intuition is just that-- if you're this close to optimal, and optimal is actually an integer, and you found an integer, then you're going to be less than 1 away, which means you're actually the same thing. OK. But let's do it more formally.

So we're within a 1 plus epsilon factor. I'm going to call the epsilon part relative error. All right. That's how much it gets multiplied by OPT in order to compute the error bound. So the relative error is epsilon. Epsilon is-- I guess is-- at most epsilon-- epsilon is 1 over 2k, which all I'm going to need is that this is strictly less than 1 over k.

So this means if I look at absolute error-- so in case you're not familiar-- relative error is I take my approximate solution-- I subtract off, let's say the optimal solution, did I get this right? This is a maximization problem. So yeah, my solution's presumably-- no, it's going to be the other way around.

For maximization problem, it's going to be-- the optimal could be bigger than me, so I take that difference-- this is called absolute error. OK. And relative error is when I just divide that by OPT. That's relative error. So I have this one part already. So usually you state it in terms of 1 plus epsilon. If you state it in terms of relative error, the 1 disappears. You just get epsilon.

The absolute error which is OPT minus APX is I take the relative error and I multiply it by OPT. OK. So relative error is going to be less than 1 if OPT is-- I guess-- greater than or equal to k? I have less than in my notes. But if OPT is greater than or equal to k, then absolute error is less than 1. Right? I hope. Let's check.

The relative error is actually OPT divided by k. Oops. No, I've got it the wrong way around. It's correct in my notes. OK. Relative error is this thing. It's going to be strictly less than OPT divided by k. This thing times OPT. OK. So as long as OPT is less than or equal to k, this thing will be strictly less than 1. That's good. OPT error less than 1 for an integer means that we actually have the same value.

I have that written down more formally. So let's go here. So if we find an integral solution-- of value-- values the objective function we're trying to maximize. Let's say we achieve something value less than or equal to k. Which it better be about if OPT is less than or equal to k. Then OPT-- OK-- this is basically doing the computation again in another way.

We had 1 plus epsilon. Epsilon's chosen to be 1/2 1 over k. And then k was the solution value that we found. And so we have this relation between OPT and the thing. And therefore-- and this works out to exactly k plus 1/2. So this is, again, strictly less than k plus 1. And so if we found-- we assumed that OPT was less than or equal to k. And so now it must actually be equal to k, because there are no integers between k and k plus 1/2. OK. I probably could have done that shorter.

So when we have an EPTAS, we exactly get an FPT algorithm. And that's-- the reverse does not hold. There are some problems that have FPT algorithms but do not have EPTASes. But, it's something, and it connects these two fields. And that's all we'll say about fixed parameter algorithms. Any final questions? Cool. See you next week.