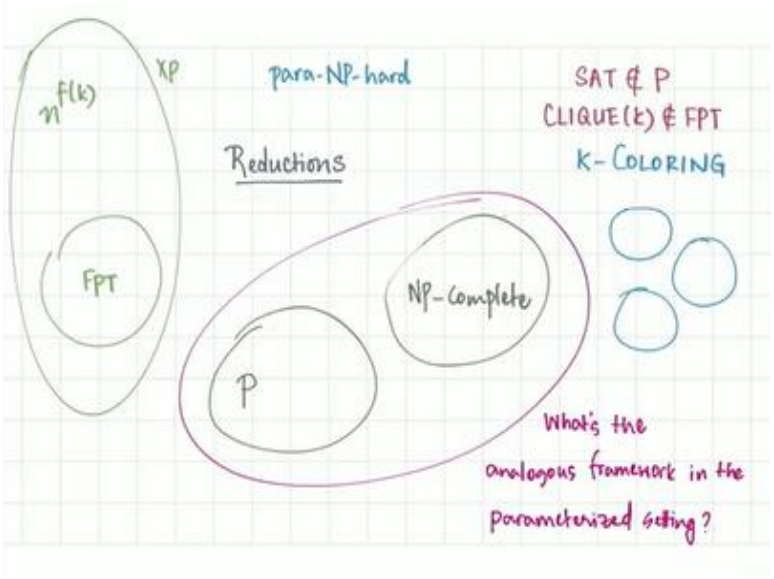
Parameterized Algorithms
Prof. Neeldhara Misra
Prof. Saket Saurabh
The Institute of Mathematical Science
Indian Institute of Technology, Gandhinagar

Lecture – 47
Reductions – An Introduction

(Refer Slide Time: 00:11)



Welcome to the 12th and the final week of the course on parameterized algorithms. So, in this week we are going to shift gears a little bit and talk about reductions and the limits of efficient computation in the parameterized framework. In other words, our discussions are going to focus on parameterized interact-ability in contrast with the work that we have been doing so far. In the first 11 weeks which has been mostly focused on coming up with efficient FPT algorithms for a wide variety of problems.

Now, in the classical setting you are already familiar with the classification of problems into classes like P, NP and NP complete and so on. And, the theory of NP completeness gives us a nice rigorous framework to talk about, when a problem is unlikely to admit an efficient polynomial time algorithm. So, the way this typically works is that, we start off by assuming that a problem like satisfiability for instance does not admit a polynomial time algorithm.

And starting with this assumption and using tools like reductions, we are able to conclude many things about many other problems. So, that is how it works in the classical setting and you might have wondered about what are the analogous notions in the parameterized world. So, first let us talk about efficient computation. What is the parameterized analogue of P? Well, we have been talking about this for about 11 weeks, so no points for guessing.

Our notion of efficient computation for a parameterized problem is fixed parameter tractability and we call this class FPT. So, I hope you will agree that FPT is a natural generalization of the idea of polynomial time in the context of classical problems, given that a parameterized problem is a classical problem along with a parameter. A natural notion of efficiency would be an algorithm whose running time becomes polynomially bounded whenever the parameters themselves are bounded say by a constant.

Now, of course when you put forth this end goal, you could also be working with the so-called XP running times, these also end up being running times that are polynomially bounded, if the parameters are bounded by a constant. But unfortunately, what happens with these so-called XP running times is that, the degree of the polynomial bound that you are working with is actually a function of k.

So, for different values of the parameter you have different polynomial bounds and that is not so exciting. While on the other hand a FPT algorithm would give you a uniform polynomial bound and the variations coming in from changes in the parameter would only be the multiplicative f of k factor. So, our notion of efficient computation in the parameterized context is going to be FPT. So, now let us talk about parameterized intractability.

Before getting into problems that are unlikely to be FPT, let us think about whether there might be parameterized problems for which even this XP running time which seems like a much easier target to work with is elusive and are there parameterized problems that, we do not even expect to be in XP. Actually, we already know such problems and let me propose one for you. Consider the problem of K colouring, which is the question of whether an input graph can be partitioned into K sets.

All of which are independent and recall that even three colouring is already NP complete. If you wanted to know, if a graph can be partitioned into three independent sets, that is a classic NP complete problem. So, if you were to parameterize K colouring by the number of colours, then notice that, if this parameterized problem did admit XP algorithm. Then in particular three colouring which is you know, basically substituting for three in the parameter K would actually admit a polynomial time algorithm.

Because, you would just run your XP algorithm with 3 plugged in for K and no matter how crazy your function f is, f of a constant still going to be some constant and this would end up being essentially a polynomial time algorithm for the three colouring problem. So, you can conclude based on just what you already assumed in the classical world that K colouring parameterized by K is not going to even admit a XP algorithm.

So, these problems are called Para- NP- hard, just to basically remind ourselves that these are NP hard or NP complete, even for constant values of the parameter. In other words, you do not even expect a XP algorithm for such problems. And, here notice that, we did not need to make any special assumptions beyond what we already make in the classical world. So, now let us move on and talk about problems that are unlikely to be in FPT.

Although they are evidently perhaps in XP. You can think about many problems for which we have not yet seen FPT algorithms. But the XP algorithms are natural brute force algorithms. So, examples would include problems like clique, dominating set, independent set and so on and so forth. So, what about these problems is it, just a matter of time before we discover FPT algorithms for them or is there something more going on here.

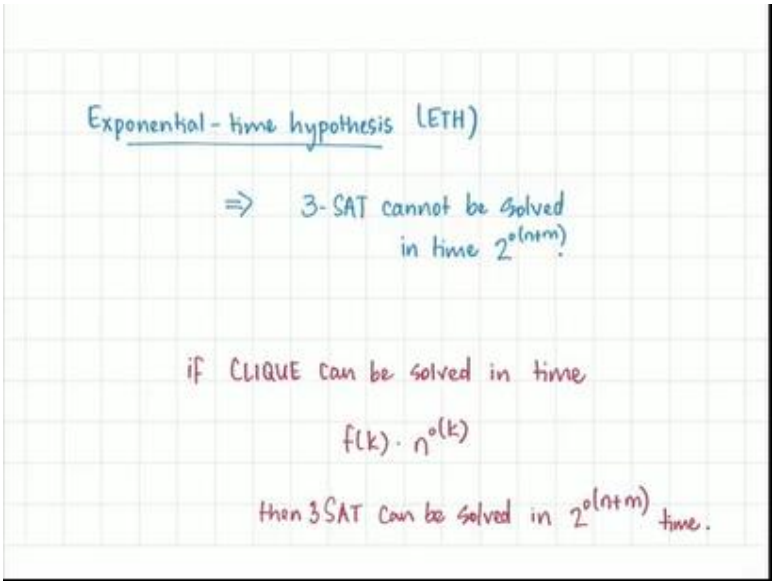
So, to be able to meaningfully talk about problems that are unlikely to be an FPT, we do need to make some additional and explicit assumptions, just like in the classical world we start off with the assumption that sat cannot be solved in polynomial time. Here we are going to work with the assumption that clique when parameterized by the standard parameter the solution size is not in FPT.

Now, this may look or feel a little bit arbitrary, if you have done your NP completeness you know that, there is a lot of justification for why satisfiability is considered to be a problem that cannot be knocked out in polynomial time. And, here this seems like we just picked a problem out of the hat, one that we were probably struggling to solve historically and we just declared. Let us just assume that this is not going to have a FPT algorithm.

So, is it just that arbitrary or is there something more that we can say? Ideally, I want to be able to say something like the following. If clique has a FPT algorithm then, SAT also has a polynomial time algorithm. If this was true then, we would be at par with the classical world in terms of the set of assumptions that we are making that forms the basis of our hardness theory. As far as I know this implication is not known to be true.

But a weaker implication is known to be true and we do have some connection with the classical world. In the sense that, if clique does admit a FPT algorithm, then some surprising things would actually happen and just to make this concrete, I would have to bring up the exponential time hypothesis.

(Refer Slide Time: 07:39)



Now, I am not going to state the exponential time hypothesis or the ETH in its original form. But, really focus on one of its most well-known implications, which is that 3-SAT does not have

an algorithm that runs in time due to the little of n + m, where n and m are respectively the number of variables and clauses in the 3-SAT formula. So, notice that this is clearly a stronger assumption compared to say just saying that 3-SAT does not have a polynomial time algorithm.

What this is saying is forget polynomial time, even if I give you much more generous resources you will still not be able to solve 3-SAT. So, if you are super conservative, you may only believe P not equal to NP and you may only believe that 3-SAT does not admit a polynomial time algorithm, but anything beyond that is fair game. And I would say that is industry standard belief, I would say most people believe this for sure.

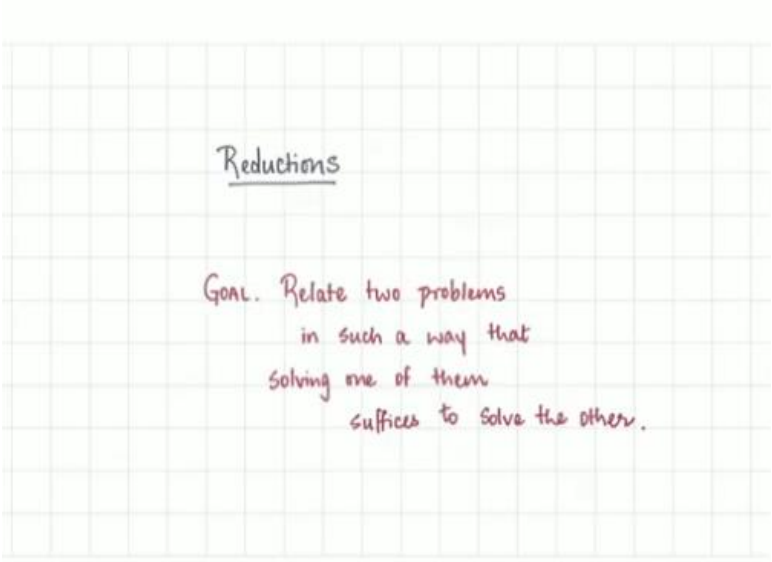
But I would also say that many people also believe in this stronger assumption, that 3-SAT does not in fact even admit an algorithm with running time 2 to the little of n + m. And if you happen to believe this as well then, you will also believe that clique when parameterized by solution size is not an FPT. The reason for that is that, we actually can show the following, if clique was an FPT with the standard parameter, then we can leverage this FPT algorithm cleverly to come up with algorithm for 3-SAT.

That runs in time 2 to the little of n + m and that in turn would violate the exponential time hypothesis. So, the long and short of it is that with a small upgrade in your baseline assumption from P not equal to NP to the exponential time hypothesis. You can lay down the groundwork for believing that clique is not an FPT with the standard parameter. So, at this point you might be wondering, what complexity class does clique belong to?

What is the parameterized complexity class that contains clique? And that is a question that we will address a little bit later. For now, what I want to do is just start with this assumption that clique is not an FPT and explore the implications of this assumption. If we know that clique is not an FPT, then what other parameterized problems can also be shown to be not an FPT based on this sort of a starting point.

So, to be able to conduct such an exploration we need to talk first about reductions which is going to be our primary tool for going between parameterized problems.

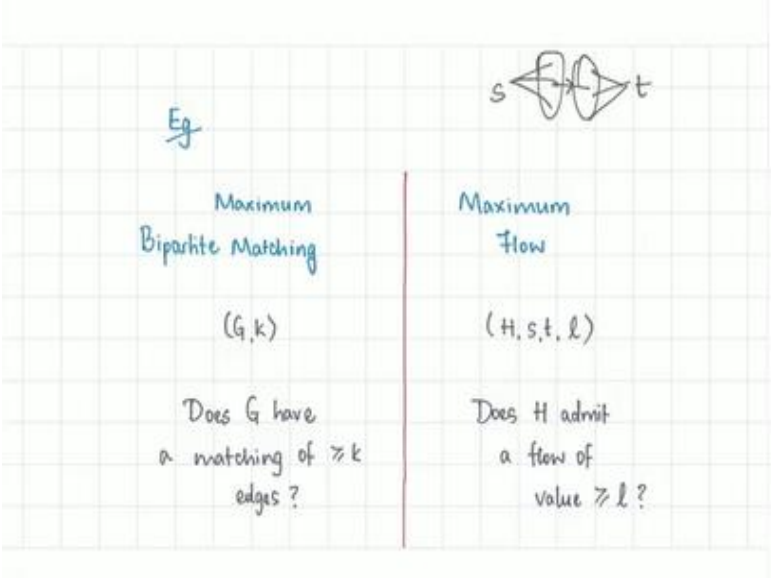
(Refer Slide Time: 10:27)



So, of course you are already familiar with the notion of a reduction. The idea is to relate two problems in a way that you can use any non-algorithms for one of the problems to solve the other problem. That is the basic genesis suffices reduction is all about. And we have already used reductions even between parameterized problems. So, if you think back to for instance, when we came up with algorithms for vertex cover with the above guarantee parameter.

We saw how those algorithms could be exploited for solving problems that did not look like they had anything to do with vertex cover or even graphs. But we were able to take those problems and morph them into instances of vertex cover above guarantee with respect to matching or NP or whatever. You have probably also used reductions quite a bit in your algorithms course.

(Refer Slide Time: 11:23)

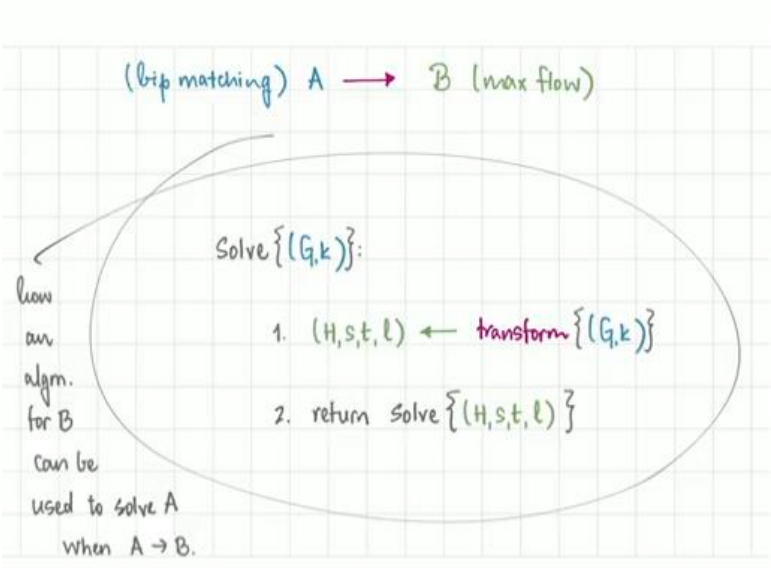


So, for instance chances are that you have learnt about an algorithm to find a maximum flow in a flow network. And you have probably used this algorithm to come up with polynomial time algorithms for a whole host of other problems. For instance, finding a maximum matching in a bipartite graph is a popular example. So, feel free to ignore the specifics of what I am talking about here just in case you have not seen max flow or if maximum bipartite matching was not a specific example that you have done before.

You cannot worry about the specifics here, but just focus on the overall mechanics of what I am talking about. So, for maximum bipartite matching in particular what you would typically do is start off with an instance of bipartite matching and then make sure that you orient all the original edges a particular way. And then you would add an artificial source in sync vertex and then you will take this modified instance and outsource it to the max flow algorithm.

Which will then do something and you can use the information that you get from this algorithm to recover a maximum matching back in the original graph. So, that is hopefully enough to trigger your memories about how this algorithm used to work. But again, if you have not seen this particular example before, do not worry about the specifics here.

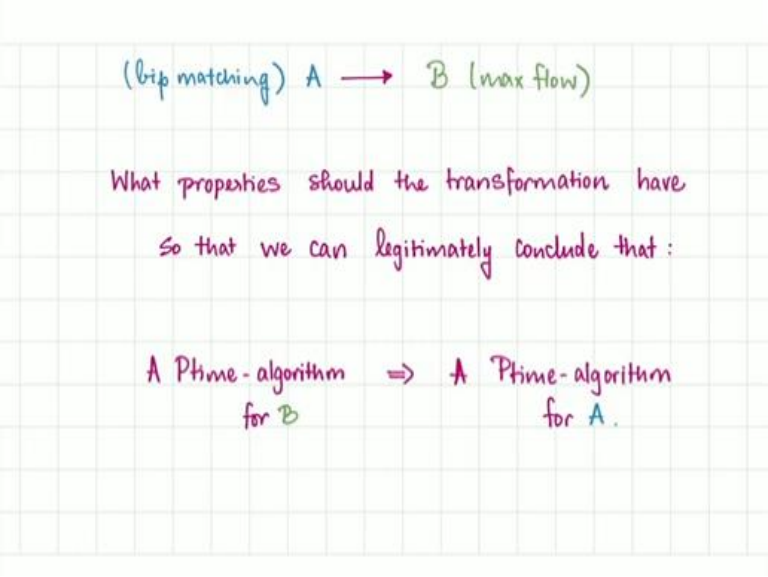
(Refer Slide Time: 12:45)



What I want to focus on is the overall way in which this reduction worked. So, in general, if you are going between two problems say A and B, so we went from maximum matching to the max flow problem. I should say maximum bipartite matching to maximum flow. And, so when we wanted to solve an instance of the source problem what we did was that we transformed it to an instance of the target problem.

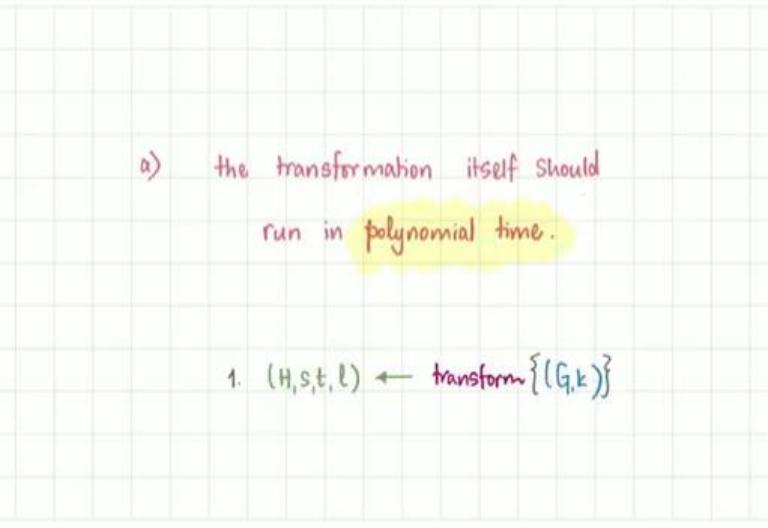
And then we ran an algorithm that we already know for the target problem and we used the outcome of that algorithm to meaningfully reason about the source problem. So, when you reduce an instance of a problem A to an instance of a problem B, what happens is that, if you know how to solve B, then you should be able to combine that algorithm with the reduction to be able to solve A.

(Refer Slide Time: 13:39)



So, just to recap here is the end goal. If we are able to reduce A to B, then what we want to be able to do is to say that, if B has an efficient algorithm, then so does A. This is what we want the reduction to empower us to say. So, given that this is what we eventually want, how should we define the concept of a reduction? So, I am just trying to reverse engineer by reductions are defined the way they are based on what we want to be able to conclude, whenever we are able to reduce one problem to another one.

(Refer Slide Time: 14:19)

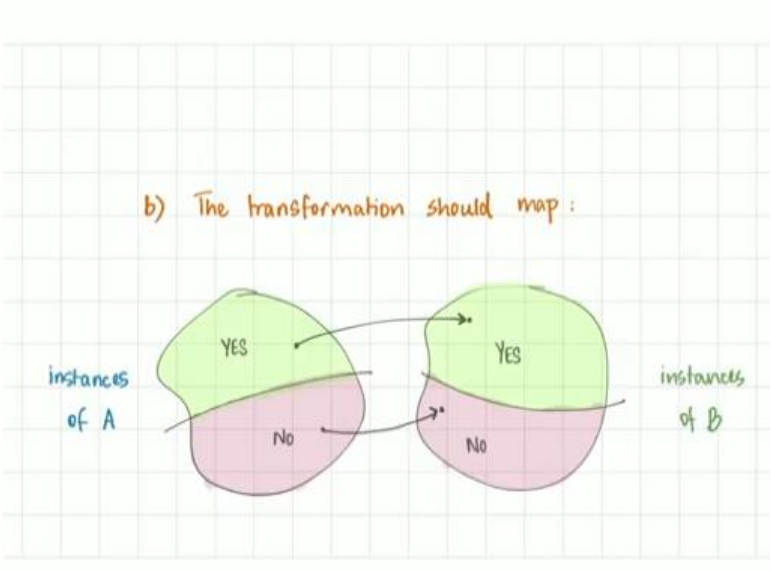


So, based on this end goal; notice that it is quite natural that we would want our reduction itself to run in polynomial time. Because remember the way the algorithm for A is going to work is perform the reduction and then solve the reduced instance. So, the first step which involves

performing the reduction, if that is already too expensive then we will not be able to use the reduction as we hope to.

So, this is the first demand that we make from our reduction that it should run in polynomial time.

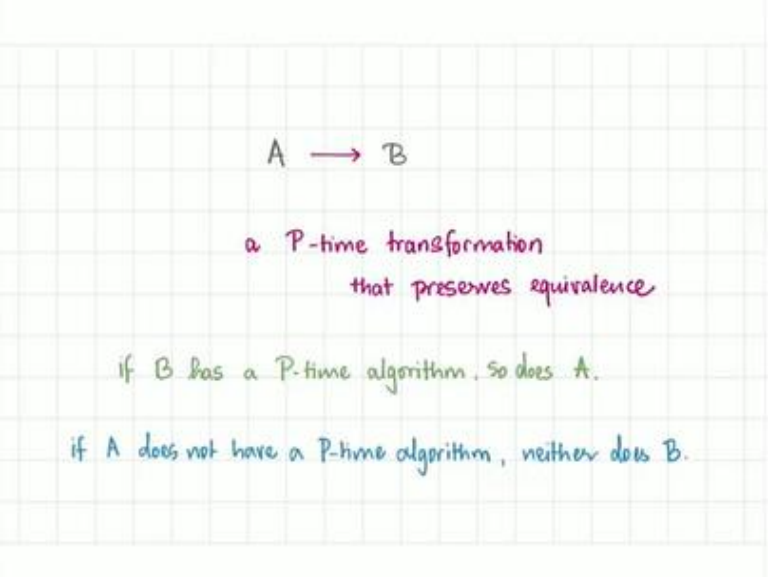
(Refer Slide Time: 14:48)



And the second thing that we want from it is more from the perspective of the correctness of the approach that we are taking. So, remember in the second step what we are going to do is solve the reduced instance of B that is generated by the reduction and essentially report that as the answer for the instance of A that we started with. For this to make sense, for this to be meaningful, we want to make sure that our reduction always maps YES instances of the problem A to YES instances of the problem B.

And NO instances of the problem A to NO instances of the problem B. This is called the equivalence property and this is absolutely critical for your plan to work. So, if this is messed up, then your reduction will simply not lead you to an efficient algorithm for A whenever you have an algorithm for B. If there is no connection between them of this sort, then the whole plan will fall through. So, this is a very crucial property of a standard reduction.

(Refer Slide Time: 15:49)

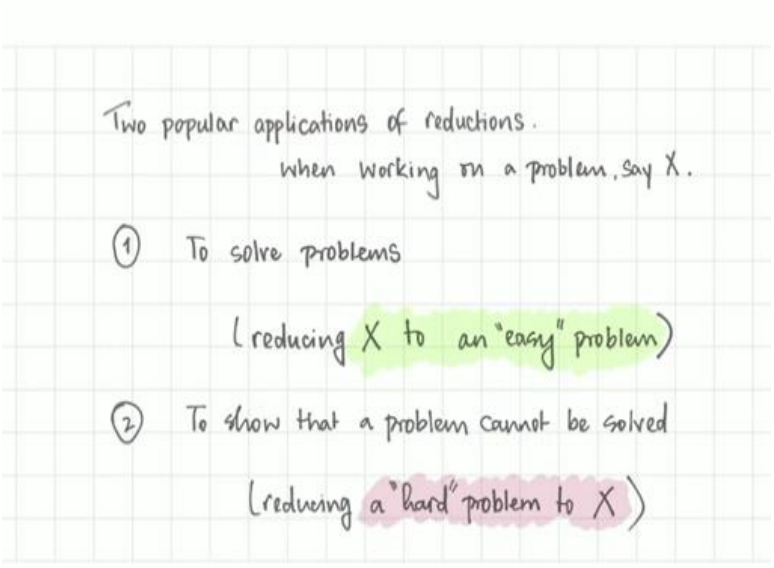


So, to summarize what we just said, you can think of a classical polynomial time reduction between two problems A and B as essentially a polynomial time transformation that preserves equivalence. And what you get out of this is the following implication? If the target problem B has a polynomial time algorithm, then so does the source problem A. And remember we get this implication by chaining together the reduction with the algorithm that we know for B.

And this amounts to being an algorithm for A. And the reason this is a valid polynomial time algorithm is because of the two properties that we just saw that the reduction has. Now, if you turn this around you can also read this statement another way and say that if for whatever reason we know that A does not have a polynomial time algorithm or you could say that if A is a problem that is unlikely to have a polynomial time algorithm, then neither does B.

So, let us see how this works? Suppose A did not have a polynomial time algorithm but B did, then just by the statement that we made a moment ago, we know that this would imply that A has polynomial time algorithm as well. But that is a contradiction to our assumption that A does not have a polynomial time algorithm.

(Refer Slide Time: 17:16)

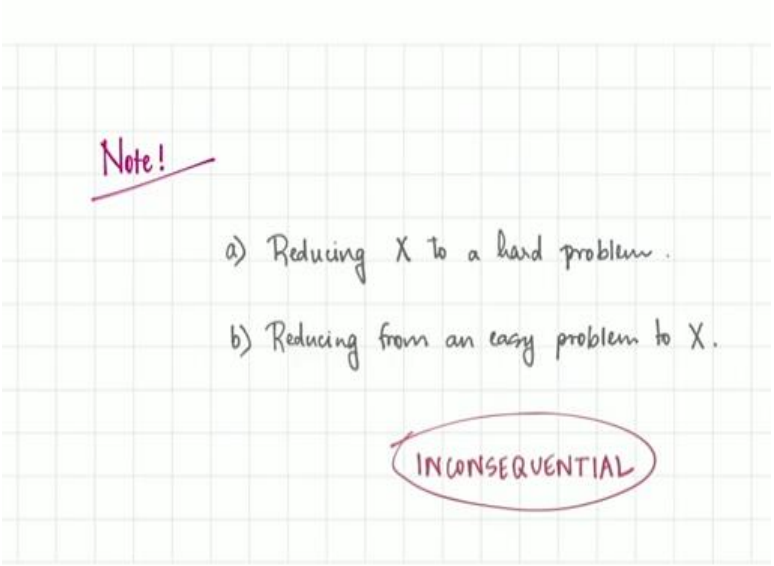


So, you can think of reductions as a double-edged sword, which can be used in one of two ways and both of these applications are quite popular. On the one hand let us say you are working on some problem; I am going to call it X. On the one hand you can reduce X to an easy problem that is what happened when we reduced say a problem to LPVC or TOMAX flow and so on. So, you can take the problem that you are working on and reduce it to something that you already know has an algorithm of the kind that you want.

On the other hand, you can also reduce from a problem that is known to be hard. So, something like satisfiability or clique or whatever. You are able to come up with a reduction from one such problem to the problem that you are working on then that makes your problem at least as notorious as the problems that you came from and that is again because of the same set of implications.

So, if somebody was able to come up with an efficient algorithm for your problem this reduction would imply that you can also derive an efficient algorithm for the known hard problem, which would generally be considered. Unlikely and therefore that would establish the status of your problem also as one of the hard problems.

(Refer Slide Time: 18:33)

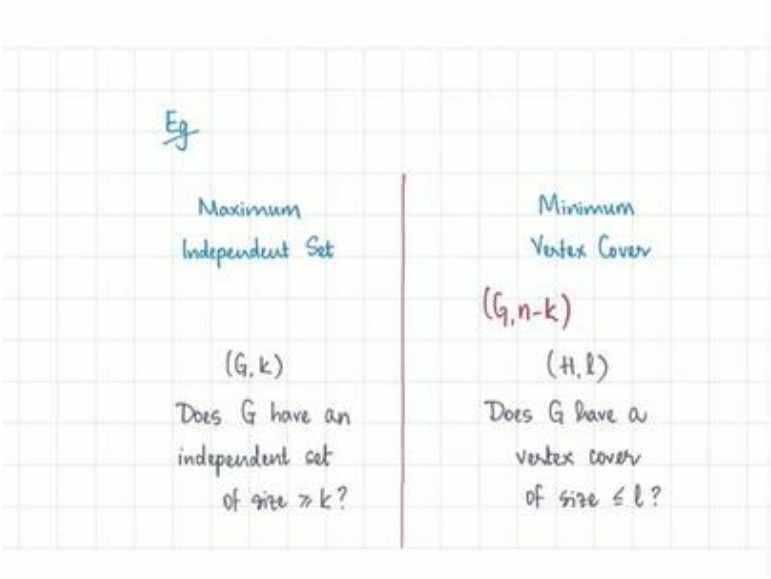


One thing to note is that, if you reduced an unknown problem to a hard problem or you reduced from an easy problem to a problem whose status you do not know anything about. Then, reductions in these directions are basically inconsequential; you do not get anything out of performing these reductions. And typically, when you are starting out and even later, sometimes the directions in which these reductions go can be a little bit confusing.

So, do not try to learn this so much by route as by just really following along with the logic of what a reduction helps you to do. So, when you reduce from A to B you can leverage algorithms for B plus the reduction to get an algorithm for A. This implies that an algorithm for B gives you an algorithm for A or if A is something that you already know to be hard, then that would imply that B is hard as well.

So, you are always either reducing from a hard problem or you are reducing to an easy problem, these are the two things that are useful to do. The other combinations would not really well. You could come up with valid reductions, but it would not really mean anything, it would not imply anything useful for you. So, this would not be a very productive thing to do. So, now let us switch gears a little bit and start talking about parameterized reductions and how they contrast with classical ones.

(Refer Slide Time: 19:59)

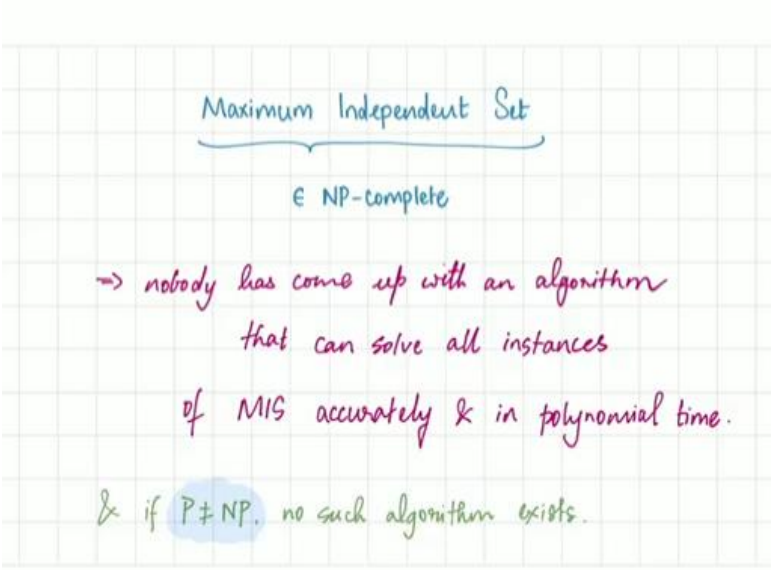


To do that, let us actually start off by talking about a classical reduction, well-known connection between maximum independent set and minimum vertex cover. So, you know that a graph has an independent set of size k, if and only if it has a vertex cover of size n - k. And this is a relationship that simply follows quite directly from the definitions of vertex covers and independent sets.

So, that naturally prompts a reduction from say maximum independent set to vertex cover. If you start with an instance of maximum independent set given by G, k then you can transform it into an equivalent instance of vertex cover by simply declaring the reduced instance to be G, n - k. So, that is just the equivalence is just what we said before about the way the definitions are related.

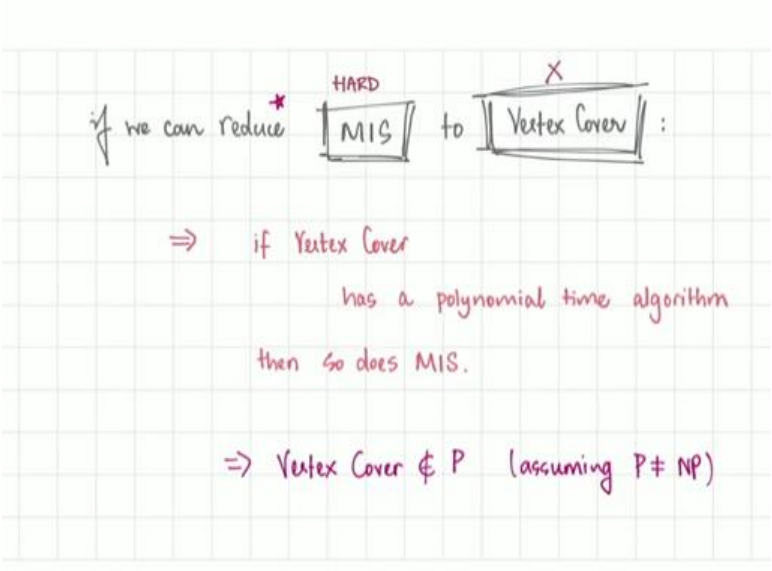
So, just to repeat that G has an independent set of size at least k, if and only if G has a vertex cover or size at most n - k, in particular if s is your independent set, then the complement of s is the vertex cover and the other way round. So, this is a perfectly valid polynomial time transformation that preserves equivalence.

(Refer Slide Time: 21:15)



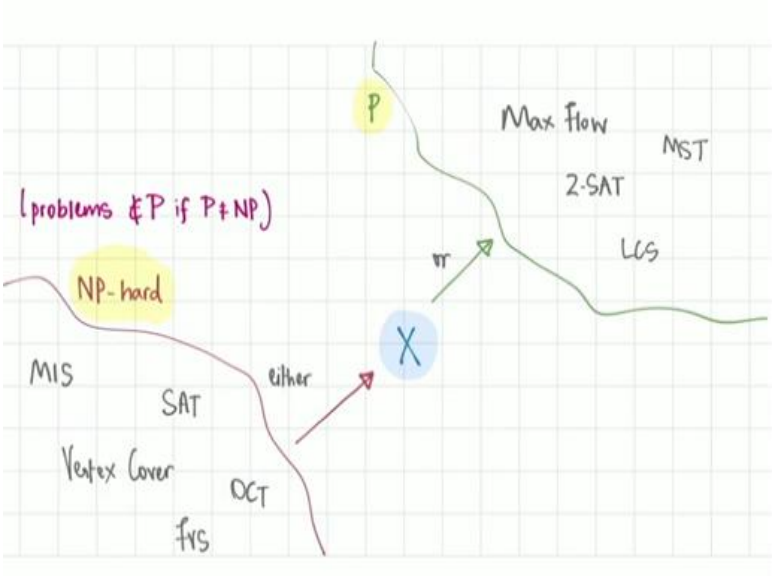
And it turns out that what is known in the classical world about independent set is that it is a NP-complete problem and in particular nobody knows a polynomial time algorithm for it, it is considered to be unlikely and if you believe P not equal to NP. Then, there is no polynomial time algorithm for independent set.

(Refer Slide Time: 21:35)



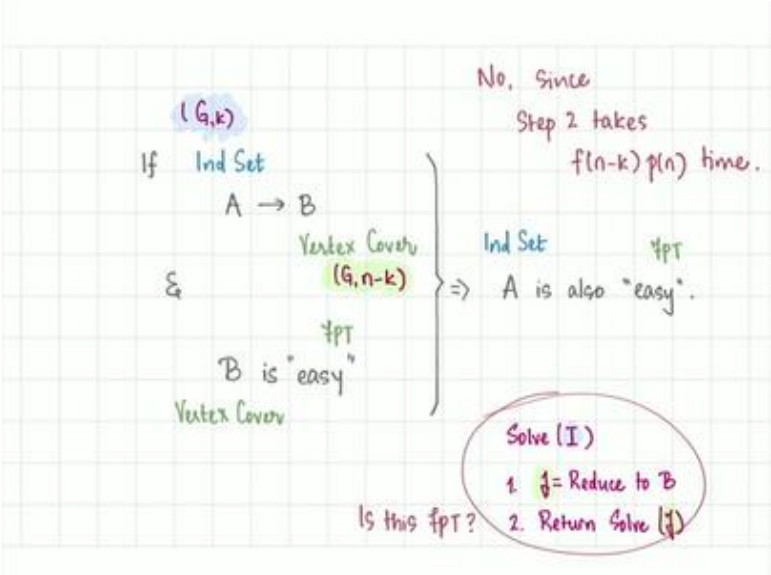
Because of the reduction what we know now is that, if vertex cover had a polynomial time algorithm, then so would independent set. So, we conclude that if we believe that P is different from NP, based on the reduction that we just had vertex cover also does not have a polynomial time algorithm.

(Refer Slide Time: 21:54)



So, once again just to recap what we have already said and I will recap this once again because it is really crucial to remember the directions of your reductions. So, when you are working with the problem X, you either reduce to a problem for which you already have an algorithm, that is useful because it helps you solve X or you start off with a problem which you know to be hard and then, you reduce that problem to the problem X and that will establish the hardness of your problem as well.

(Refer Slide Time: 22:26)



So, one of the things that is going on here is that, you know we have said this quite a few times by now, if you can reduce A to B and B is easy. Then, that means that A is also easy and in the

classical world our notion of easy has been the problems admitting a polynomial time algorithm. In the parameterized world our notion of easy has been FPT. Now, we just said that we have a reduction from independent set to vertex cover.

And we also know that vertex cover is a very familiar easy problem in the sense that it is probably one of the first problems for which we saw FPT algorithms and we have been improving these algorithms ever since. So, it might be really tempting to plug in our generic reasoning here and say that well because, vertex covers FPT and we can reduce independent set to vertex cover, independent set also must be FPT.

But you know what they say sometimes about the table being in the details. So, let us just take a closer look at whether this argument actually pans out, let us fill in the details and see what is actually going on here. So, remember if you start off with an instance of independent set which is given by G, k the reduction is going to produce an instance of vertex cover which looks like G, n - k. Now, how do we get this FPT algorithm for independent set?

Remember our recipe is first run the reduction and then run the known algorithm for the target problem on the reduced instance. So, let us do that, let us perform these two steps. We first perform the reduction that is no problem, that is polynomial time and we have very little work to do there we just have to compute n - k, so that we know what sort of a vertex cover we are looking for.

In the next step crucially, we have to run the FPT algorithm for vertex cover on this so-called reduced instance and then report the answer, as the answer for independent set. So, because these instances are equivalent there is going to be no problem with the reporting, this is a perfectly correct algorithm. But the thing we really have to think about is whether this is a FPT algorithm in k, which is the original parameter that we are interested in.

So, really pause this for a moment here, especially if you are seeing this for the first time and think about whether you are willing to commit to this whole process being a FPT algorithm or not. So, I hope you had a chance to think about that. So, the instance G, n - k when subjected to

your favourite vertex cover FPT algorithm, let us say we just do the simple two-way branching which runs in time 2 to the k.

It is 2 to the k where k is the parameter of the vertex cover instance and this parameter right now is actually n - k. So, when you run the FPT algorithm you will end up spending time which is not really FPT. It is FPT in the given parameter, but the given parameter is not a function of the original parameter. It is all messed up, it is n - k. So, your vertex cover algorithm will run in time 2 to the n - k, which is no good as far as independent set is concerned.

This is not FPT and k it is not FPT in the original parameter. So, this motivates us to take a closer look at the definitions of reductions in the parameterized context. It is clear that, if we just continue working with polynomial time reductions as we have seen and known them so far. They may not really give us the implications that we want these definitions are not nuanced enough for the parameterized context.

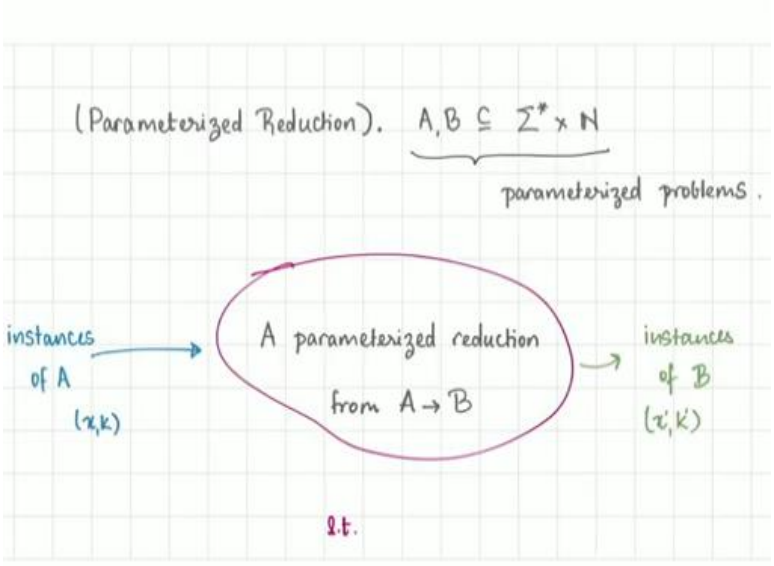
And we cannot really blame them, because those reductions were defined in the context of classical problems. There was no parameter in the scene to worry about. But now we need to really nudge these definitions a bit to account for the presence of these parameters. And we need to define things in such a way that again just have this reverse engineering mindset, because we know that what we want from reductions is the following.

If you do have a reduction from A to B and B is FPT, you want to be able to conclude that A is FPT. And generally speaking, you even know how you will go about showing this. The way you will come up with the FPT algorithm for A, based on the FPT algorithm for B and the reduction is the following just like with classical reductions what you would want to do is run the reduction first and then run the FPT algorithm for B, on the reduced instance that is generated by the reduction.

And what you want is this two-step process to be FPT overall in the original parameter. So, you know what you want to prove based on the definition of a reduction. You even know how you

are going to prove it? So, all that remains is to fill in the details in the definition of the reduction, so that things work out according to plan.

(Refer Slide Time: 27:23)

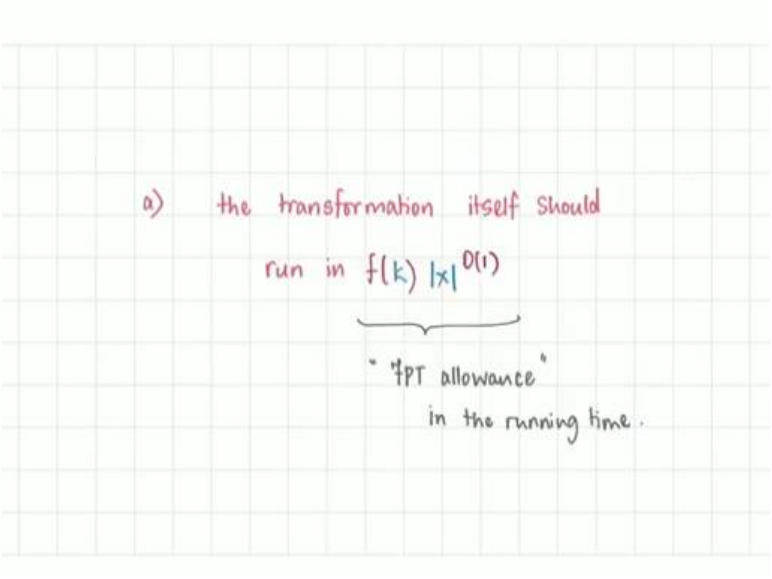


So, let us say that we have parameterized problems A and B and as I have been saying we want to define an appropriate notion of a parameterized reduction. So, that whatever we said goes through. If you like take a moment here and try to come up with this definition on your own and you can come back whenever you are ready, so we can exchange notes. So, some things never change.

So, in the classical setting we wanted the reductions to preserve equivalence and this continues to be the heart of the matter even in the parameterized setting. If you are not mapping YES instances to YES instances and NO instances to NO instances your reduction is any way broken. So, that aspect remains the same, but what we have to take a more nuanced approach on is how is the parameter transforming?

As we go from the original instance to the reduced instance, because that will play a key role in determining the running time of the overall two-step process that we have been proposing. So, let us take a look at the definition of a parameterized reduction. Here is the textbook definition and it should feel very natural given all the lead up that we have talked about so far.

(Refer Slide Time: 28:39)

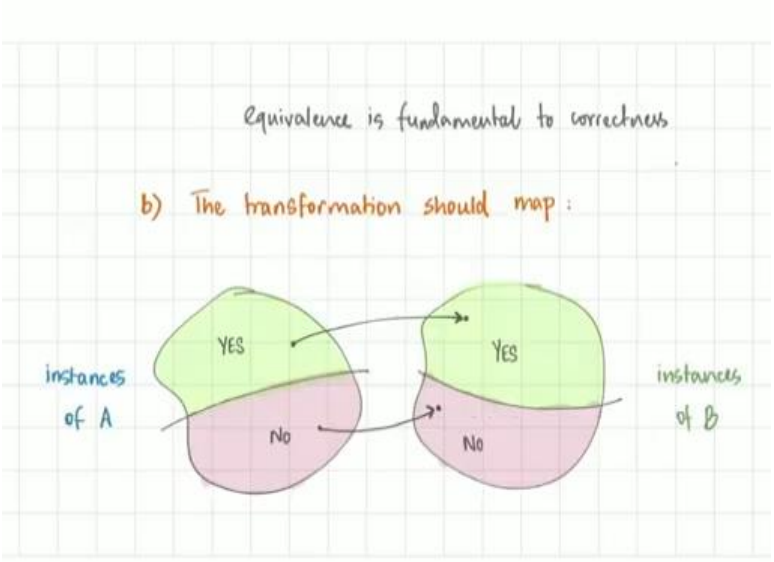


So, the first thing that we want out of a parameterized reduction is that the reduction itself should run in time that is FPT in the parameter of the source instance or the instance that you are reducing from. The reason for this is that well, this is the first step of your ultimate algorithm for problem A. So, you are going to run the reduction and then you will do whatever. So, if this first step itself is doomed then your plan will fail.

So, your parameterized reduction itself should run in FPT time. Notice that this is actually more general, it is more allowance compared to the classical analogue. In the classical setting where polynomial time reduction is only allowed polynomial time. Now you are actually allowing yourself more time and for this reason in fact not every parameterized reduction is a valid polynomial time reduction.

Because it may be more expensive than what is permitted in the classical world. So, this is the first property that we want, it is an analogue of the first property that we demanded from a classical reduction, where we said the reduction should run in polynomial time, now a natural generalization of that is that the reduction should run in FPT time.

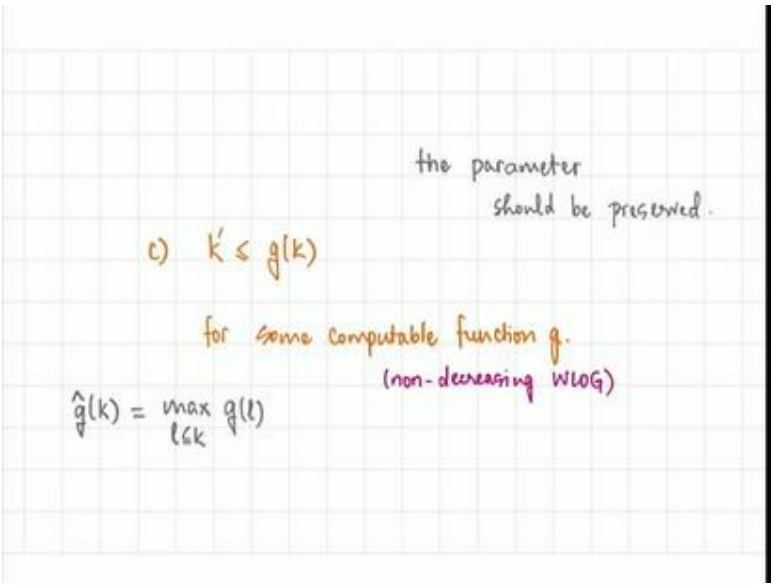
(Refer Slide Time: 29:51)



Now, the second demand that we make from parameterized reductions is that they should preserve the equivalence of the instances that they are working with the ones that they start with and the ones that they generate and as we have already mentioned this is. Basically, the same as our analogous expectation from a classical reduction and is absolutely fundamental to the correctness of our overall approach for coming up with a FPT algorithm for A, based on the reduction from A to B and an FPT algorithm for B.

For that whole idea to work it is absolutely crucial that the reduction has this property. So, just to really spell it out, if the reduction maps an instance X, k of A to some instance Y, k prime of B, we would want that X, k is a YES instance of A if and only if Y, k prime is a YES instance of B. So, this is equivalence and this is the aspect of the definition that will help us argue the correctness of our, would be algorithm for A.

(Refer Slide Time: 30:54)



Now, let us turn to the final property which will help us argue that, the overall algorithm that we are going to propose for A, actually runs in FPT time and is a valid FPT algorithm for the problem A. So, what we want is that the parameter of the reduced instance is bounded as some function, some computable function of the parameter of the source instance that we are starting with.

We will see how this will be useful later, in arguing that our approach actually leads to a FPT algorithm. This is a small technical remark here, which you do not have to worry about so much, but we are going to say that this function that establishes the bound-on k prime is a non-decreasing function. You could either demand this in the definition itself or you could actually say that this is without loss of generality.

So, why is that? Well, suppose g is some arbitrary function, which is not necessarily non-decreasing. You can use g to define this other auxiliary function which I am calling g hat over here. So, you can define g hat of k to simply be the largest value of g that you have seen so far. So, let us just say that this is maximum of g of l, l being at most k. And notice that if k prime is bounded by g of k, then it is of course also bounded by g hat of k.

Which additionally has the property that it is non-decreasing and this is just a useful property to have and we will just stash this away at the back of our minds and invoke it when it is necessary

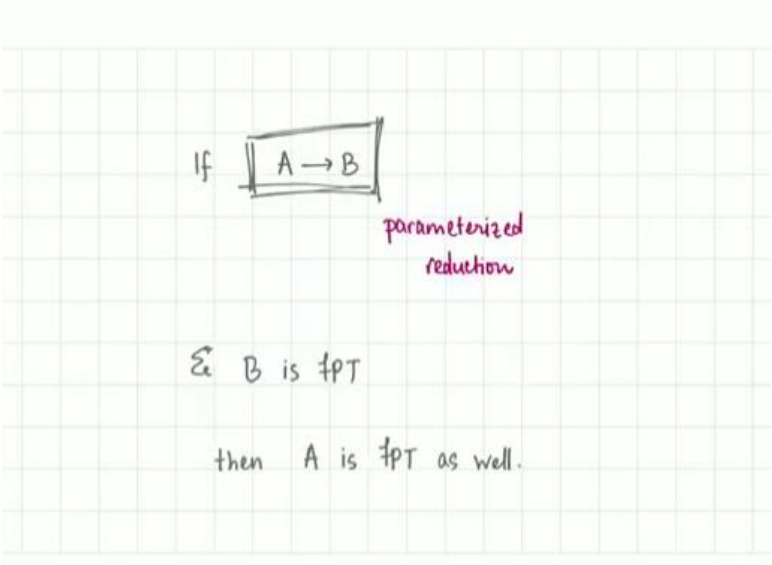
in a couple of minutes. So, with this we conclude the definition of a parameterized reduction. Just to recap a parameterized reduction is a FPT transformation between two FPT problems. So, by FPT transformation I mean a transformation that runs an FPT time, FPT with respect to the original parameter.

It preserves the equivalence and additionally it makes sure that the parameter of the reduced instance is bounded as a pure computable function of the parameter of the original instance. And, this is the key property that again sets it apart from a classical reduction. So, this is why not every classical reduction is a parameterized reduction. So, we have already pointed out earlier that, it is not automatic that a parameterized reduction should be a classical reduction.

Because it may actually end up taking more time than what is allowed in the classical setting. But now because of this last property and we have already seen an example of this with independent set and vertex cover. But this last property makes it explicit and is the reason why not every classical reduction is a parameterized reduction, because you do not expect the classical reduction to necessarily preserve the parameter value in this way.

So, now having done all the work of putting the definition together, let us confirm that we actually get what we want from it.

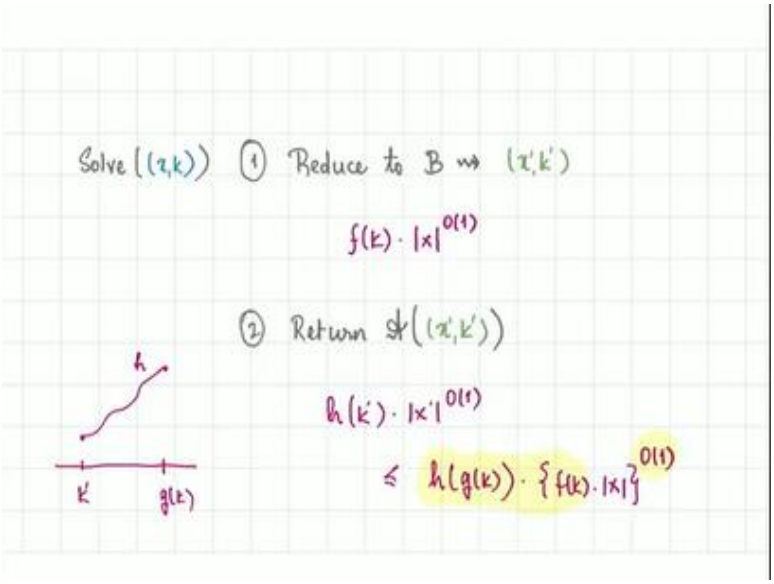
(Refer Slide Time: 33:42)



So, here is what we wanted all along, we wanted to be able to say that, if there is a parameterized reduction from a parameterized problem A to a parameterized problem B. Then, if B is FPT, then we also conclude that A is FPT. And given that we set up the definition, so that we are going to be able to say this, it is not surprising that this is true and you know the proof also is going to follow a predictable path.

But it is a good idea to just confirm that everything indeed works out exactly as you would expect. So, let us go ahead and take a look at our proof of this statement, which is what we had been perhaps waiting for all along, while we were setting up the definition.

(Refer Slide Time: 34:26)



So, here is the proposal for how we are going to get a FPT algorithm for A. So, let us say that x, k is an instance of A and here is how we are going to solve it using the two ingredients that have been given to us. The first being the reduction from A to B and the second is the presumed FPT algorithm for B. So, in the first step what we are going to do is reduce this instance x, k to an instance of B using the reduction of course.

And let us say that what we get out of the reduction is an instance x prime k prime. What we are going to do after that is run the FPT algorithm, that we know exists for B by assumption. We are going to run that algorithm on x prime and k prime and report whatever that algorithm tells us as the output of the algorithm, that is solving x, k. And already from here you can see that this

would work in terms of correctness because of the equivalence of x, k and x prime, k prime and the presumed correctness of the algorithm that solves instances of B.

So, this algorithm is correct and that is not a problem, but what we need to take a closer look at is the overall running time. So, let us try and understand how much time these two steps will take? The first step by the first property of a parameterized reduction will run in time FPT and k because, that was our first demand from a parameterized reduction that, the reduction itself should run in FPT time.

So, that is how much time we are going to spend on the first step. Second step, we are running an algorithm which is FPT for the problem B and in particular this running time is going to be FPT in k prime because, that is the parameter of the B instance that we are working with. So, it is going to have some running time of this form. It is bounded by some function of k prime and then there is a multiplicative factor.

That is polynomial in the length of x prime, which is the length of the input of the instance that we are working with. So, that is the running time in the second step and right now in this form it is not super helpful because we need to understand this running time in terms of k rather than k prime. Now, this is where we can leverage the last property that we introduced of parameterized reductions, which was parameter preservation.

Remember that this is what relates k prime and k, we know in particular that k prime is bounded as some computable and non-decreasing function only of k and we call this g of k. So, I know that k prime is at most g of k and from this I want to be able to conclude that h of k prime can also be bounded as h of g of k. And this would be true if we knew that h itself was also a non-decreasing function.

And notice that, we can actually assume that h is a non-decreasing function again without loss of generality, pretty much using the same argument that we applied to g a few minutes ago. So, given that k prime is bounded by g of k and h is a non-decreasing function, we can actually

bound h of k prime by h of g of k. And now again let us turn to size of x prime we want to understand size of x prime in terms of size of x.

So, that we; can say something meaningful about the overall running time being FPT for the original problem. But how did you generate the instance x prime? You generated it using the reduction; the reduction itself only had f of k times poly in size of x amount of time on the clock to run. So, using this much time you cannot generate an instance that is longer than the number of units of time that you had available to you.

So, this is a pretty standard argument that we often make and this leads us to the fact that size of x prime is also bounded as f of k times, some polynomial in the size of x. And this is a very loose bound I think in practice, but it works. And if you combine everything therefore you can see that the overall running time that you have is in fact a perfectly good FPT running time for the original instance.

And we have already made our remarks about correctness based on the equivalence property of the parameterized reduction. So, you can see how all the three properties that we stipulated in the definition come into play. So, the first and the third properties about the reduction running in FPT time and being parameter preserving, in the sense that the parameter of the new instance is bounded as some computable function of the parameter of the old instance alone and nothing else.

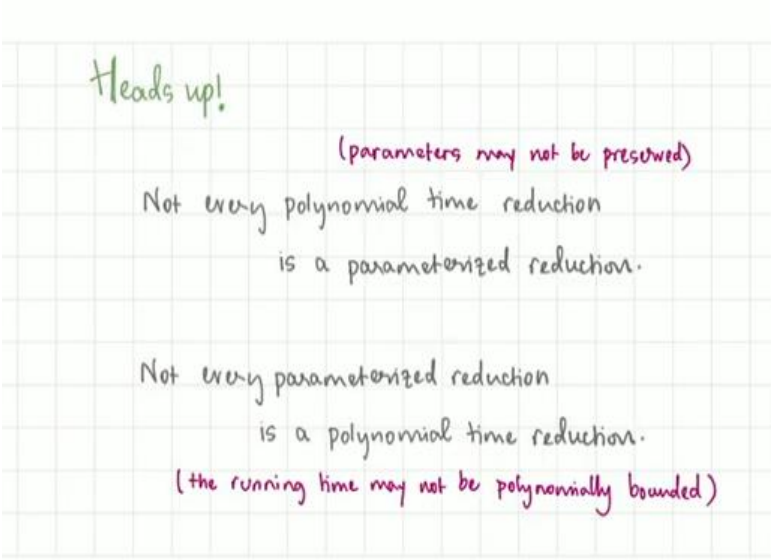
These two aspects helped us bound the overall running time of this process as a FPT function of the original parameter. And the second property which was the property of equivalence helps us argue. That this overall algorithm that we; are proposing for the problem A is in fact correct. So, that is the whole idea of a parameterized reduction once again one of the main take a ways is that not every classical reduction is a parameterized reduction.

And you will find lots of examples of classical deductions that are not valid parameterized reductions, so definitely watch out for that. And on the other hand, not every FPT reduction or not every parameterized reduction is a classical reduction. Although, here examples of FPT

reductions that are not valid classical reductions are relatively rare. We will see one such example of a reduction that really leverages the full power of this definition.

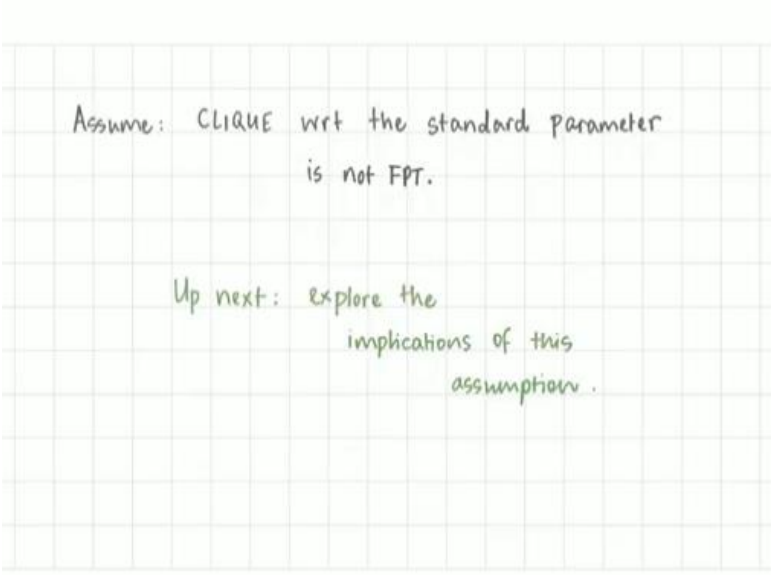
But often it is just the case that, if you come up with a parameterized reduction, it would also typically be a valid classical reduction if you were to. You know basically perform the reduction analogously for the corresponding classical instances.

(Refer Slide Time: 40:13)



So, this is just a recap of the two things that we just pointed out, it is useful to keep this at the back of your minds as we go along. So, where do we go from here?

(Refer Slide Time: 40:23)



What we are going to do next is really look at lots of examples of parameterized reductions and we will typically be starting from clique or problems that we have already reduced clique to. So, that is what is coming up in the rest of this module, we are going to look at examples of productions and how they work. And then we will come back and talk about the complexity class that clique is contained in.

This is a complexity class called W 1 and the one may suggest that may also be a W 2 and so on, which is indeed the case. So, this is another interesting contrast with classical problems, where typically all the problems that you know that are hard like independent set dominating set vertex cover etcetera. are all bundled into sort of one big class of NP-complete problems. Here, we will actually see that even within the class of problems that are hard in the parameterized setting.

There is a little more nuance to what is going on and we will see that different problems actually are hard to different degrees. And we will talk about that a little bit and basically look at some definitions, which explain these degrees and from there we will continue to look at a couple of more examples before calling it a wrap. So, that is the agenda for the rest of this week. I hope to see you back in the next segment of this module.

Where we will actually work with concrete examples of reductions, which is always an exciting thing to do. So, I am looking forward to seeing you there. Thanks for tuning in.