

**Parameterized Algorithms**  
**Prof. Neeldhara Misra, Saket Saurabh**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology-Gandhinagar**

**Lecture-16**

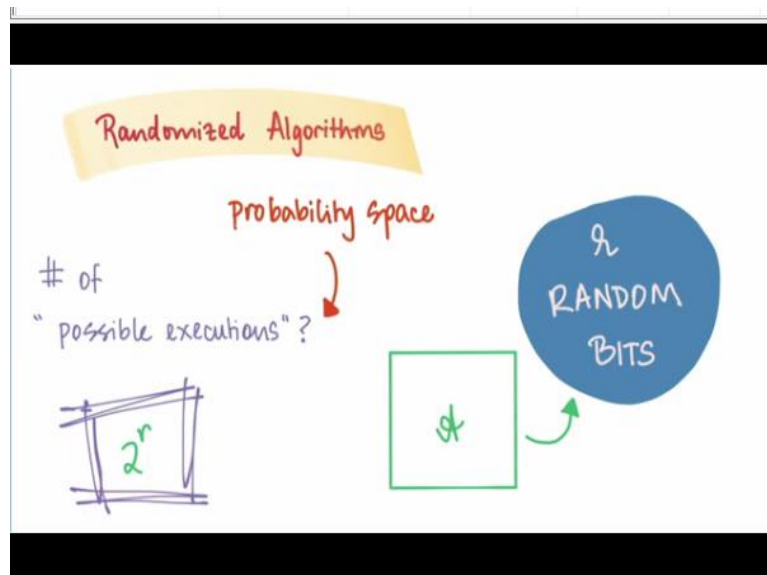
**Introduction to Randomized Algorithms via a Simple Randomized FPT Algorithms for FVS**

Welcome to week 4 of parameterized algorithms. In this week we are going to be talking about how we can leverage the power of randomness to design some really cool and elegant FPT algorithms for some of our favorite NP complete problems, some of which you have seen multiple times in this course already and some of which are going to be new and we will be meeting them for the first time at least in this course.

So, if you have never worked with randomized algorithms before then I would recommend checking out some of the links in the description of this video just to generally get a sense of how they work. Of course we will be starting with a quick overview with what it means for an algorithm to be randomized and how we are going to analyze them, but it is not going to be the big picture, it is going to be just about what we need to be able to follow the methods that we will be describing this week.

So, for just watching these videos this introduction should really be enough, but if you do want a slightly more comprehensive introduction to how randomized algorithms work and how they are typically analyzed then please do explore a little bit further by looking up the links in the description of this video. So, with that said let us get started here. So, first of all what do we mean by a randomized algorithm.

**(Refer Slide Time: 01:36)**



So, we are not going to be talking about what it means for some event to be truly random or how do people generate randomness in the real world and so on. These are fascinating discussions on their own right but unfortunately they are well beyond the scope of this lecture. So, once again if you are interested I would urge you to actually explore these topics more fully. For now what we are going to assume is that our algorithm has free access to a random bit generator which is just truly random it gives you fair coin tosses at will.

And let us say our algorithm happens to toss our such coins during its execution, it is designed to work with  $r$  of these coin tosses. Notice that once a coin is flipped or one of these random bits is drawn however you prefer to visualize this. The algorithm can choose to do different things based on the outcome. So, it might say that if the coin turns out to be heads then I am going to do this.

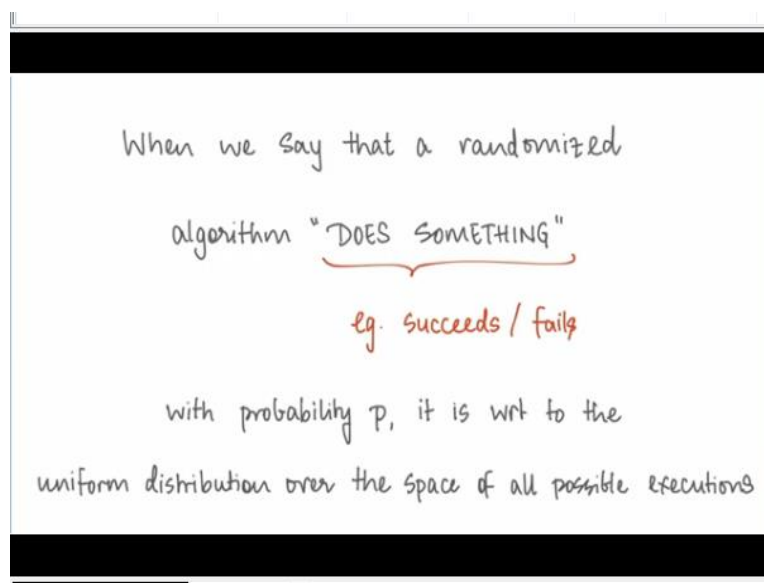
But if it turns out to be tails then I am going to do this other thing. Given that the algorithm tosses  $r$  of these coins how many possible executions can the algorithm experience on a fixed input, in how many ways can the algorithm manifest based on all the possible outcomes that these coin tosses can have? This is not a trick question but if the answer is not clear then please feel free to pause here and just think about this for a moment.

For instance if the algorithm was allowed just one random bit at some point during its execution and let us say that it used that random bit to decide 1 of 2 ways of proceeding further. Then clearly you can imagine that because of the presence of this one random bit there are 2 possible executions that this algorithm could experience on any fixed input. With  $r$

random bits just generalizing this further you can probably conclude that the number of possible executions is going to be  $2^r$ .

So, you can think of this as a space of all possible outcomes, you can think of this as the sample space of an experiment, the experiment being your algorithm and all the probability in statements that we make in the context of our randomized algorithms, all of them are with respect to this probability space. This is what we are going to be talking about.

(Refer Slide Time: 04:14)



So, typically we will say things like  $r$  algorithm does something with probability  $P$  and the something that the algorithm does is usually with reference to some task that we either wanted to accomplish or some situation that we wanted to avoid and if it is a task that we would like for it to accomplish then we think of this as a success probability that we want a lower bound. We usually want to say that it manages to do what we wanted it to do with probability at least  $P$  for some  $P$ .

On the other hand if it is an error probability in the context of something that we are worried about then we want to come up with an upper bound, we want to say that the probability of this bad event is at most something. But the point of this discussion here is just to emphasize that all of these events are being spoken of in the context of the sample space that we described earlier which is all possible runs of the algorithm given the set of random bits that it is going to be flipping through its course.

So, that is the context in which all of these probabilistic statements are being made. Now let us turn to a specific sort of a scenario.

**(Refer Slide Time: 05:30)**



Suppose I give you a randomized algorithm which has the following behaviour. So, if it gets a NO instance of the problem that it is trying to solve as input then it is guaranteed to output NO, it never makes a mistake on a NO instance. On the other hand if it gets his input a YES instance of the problem that it is trying to solve then it may or may not be able to get it right. But it does get it right with probability at least  $P$ .

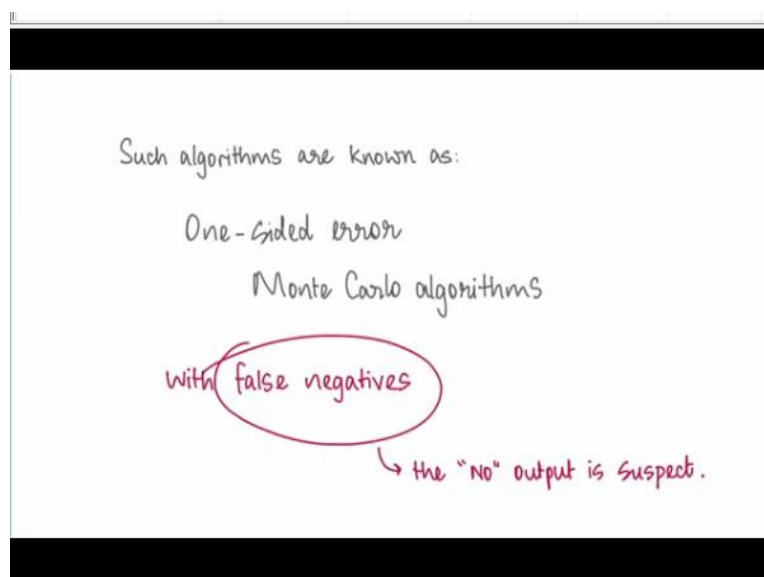
So, if it is dealing with a YES instance then it says YES with probability at least  $P$  and if the input is a No instance then it is guaranteed to say NO. So, you can think of that as an event that happens with probability 1. So, suppose you run this algorithm on some instance of whatever problem it is trying to solve and suppose the output is YES should you be suspicious about this output or can you be sure that it is right.

On the other hand suppose the output is NO again can you be sure that this output is right or should you be worried. So, in one of the cases the answer is that if the output is  $x$  then you do not have to worry you can conclude that the algorithm's done its job correctly. For the other output though you do have to be a little bit concerned about whether something was a mess. So, again pause here to think about what the correct answer should be and we will discuss this in just a moment.

So, it turns out that the output that you should be worried about is when the algorithm says NO. This can feel a bit counter-intuitive because we just said that the algorithm is good for NO instances it does the right thing on NO instances. But that is exactly why you can be sure that when the algorithm says YES it is not making a mistake, because if it said YES and it was making a mistake that means that the output is YES although the input was in NO instance.

But that is not possible because the algorithm never makes a mistake when it is dealing with a NO instance. So, whenever it says YES you can be absolutely sure that it is done the right thing. However, if it says NO you are not sure about whether this was because it was truly dealing with a NO instance or because it was dealing with a YES instance but it made a mistake with probability say at most  $1 - P$ .

**(Refer Slide Time: 08:17)**



So, such algorithms are called Monte Carlo algorithms with one-sided error and false negatives. So, the way to remember the phrase false negatives is that the NO answers could be wrong, that is how you think of it. Even though it can be again a bit confusing because it is an algorithm that performs well on NO instances that is the property but that also means that it is the NO output that you have to be suspicious about.

And that is where the phrase false negatives come from. So, it helps you remember that whenever the algorithm says NO you have to scratch your head and think about whether what you had was truly a NO instance or not. You can be sure that it is probably okay with

probability  $P$  but you still cannot be completely sure. Unlike when the algorithm says YES things are fine.

That is partly why we also use the phrase one-sided error because you have a problem with only one of the 2 possible outputs. Now there are of course algorithms that have one-sided errors going the other way with false positives and there are algorithms where you might even have 2-sided error where you might make mistakes on both outputs but hopefully with bounded error probabilities both ways.

But for the most part the algorithms that we are going to be discussing this week are going to be of this style where you can be confident about the YES outputs when you say YES you have found the thing that you are looking for sure, but whenever you say NO there may have been a missed opportunity and that is what requires further analysis. That is going to be the typical style of all the algorithms that we are going to see.

Now suppose you do have a randomized algorithm maybe something that you came up with for your favourite problem which is of this kind it is a one-sided error Monte Carlo algorithm with false negatives with a success probability of at least  $P$  but let us say that it is a success probability that you are not very impressed with and it does not give you a lot of confidence. So, you are thinking about whether you can improve the success probability.

And that is a very natural question to ask yourself, it is just like with traditional algorithm design, we keep thinking about can we do better, can we improve the running time etcetera. With randomized algorithms a very natural goal is to see if you can improve the chances of success. Now there are 2 ways that you can approach this question. One is to go back to the drawing board and try to improve the design of your algorithm.

So, that you have a better success probability, but it turns out that there is a more general way of boosting the success probability of any randomized algorithm that has a non-trivial success probability to begin with. So, in some sense you can start off with the algorithm that you have but you can bootstrap it just by simple repetition. So, let us think about this a bit, why does repetition help?

Remember you are dealing with these algorithms that have one-sided error with false negatives which again means that whenever the output is YES you can be completely confident about it but if the output is NO then that is when things may be a bit shady. So, let us say you run the algorithm once if the output is YES of course it is your lucky day and the story ends here you do not have to do anything.

But if the output is NO then you are a little bit worried, so you say okay, let us run the algorithm one more time just to be sure. So, you run the algorithm again and suppose this time on the same input the output becomes YES then you say okay, so the last run was an error but at least we fixed it this time and then again you can go home. But suppose the output is again NO then you are still not very convinced so you run the algorithm again and it is the same story.

If the answer is YES then you stop but if the answer is still NO then you may want to run the algorithm again and suppose you keep doing this and you run the algorithm say a 1000 times and it says NO every single time. At some point you start having an increasing confidence in the output of the algorithm, you start believing that maybe the reason this output is no is because it really was a NO instance to begin with because we have tried so many times. So, let us just try and formalize what is going on with this idea here.

**(Refer Slide Time: 12:52)**

Boosting the chances of success.

Success probability  $\geq p$

Repeat the experiment  $t$  times

Return NO only if all  $t$  runs say NO

$\Pr(\text{failure}) = \Pr(\text{every run is a failure}) \leq$

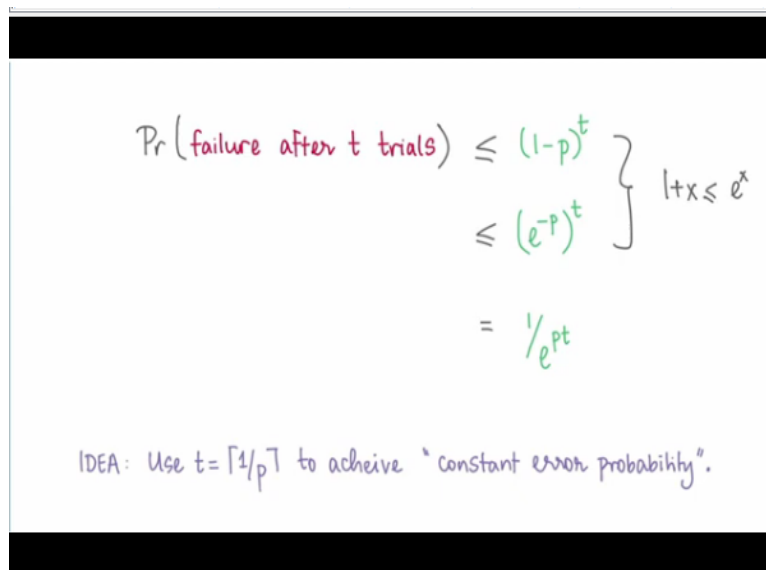
So, remember we have a base algorithm which has a non-trivial success probability and that is at least  $P$  and let us say we repeat this algorithm  $t$  times and our overall algorithm is that we return NO only if each of these  $t$  runs of our base algorithm reported NO. But at any point if

we saw a YES outcome then of course we report YES and we do that with full confidence. Now what is the probability that if you do end up saying NO at the very end you still made a mistake?

So, let us think about when that can happen? That can happen if the algorithm made a mistake in each one of these  $t$  trials. So, if the algorithm made a mistake overall that means that it must have been that we had a YES instance to begin with but every time that we ran the algorithm we just fell into bad luck and we said NO each and every time. So, what is the probability that this happens?

If you need to take a moment here just to figure this out for yourself but hopefully you will arrive at the same conclusion that I am going to show you here which is that the error probability is going to be at most  $1 - P$  to the  $t$  because each individual run fails with probability at most  $1 - P$ , remember that the success probability was at least  $P$ . So, the failure probability is at most  $1 - P$ . And for the whole algorithm to fail this failure must happen in every single run. So, it is an end of  $t$  bad events, so the overall probability is at most  $1 - P$  to the  $t$ .

(Refer Slide Time: 14:31)


$$\begin{aligned} \Pr(\text{failure after } t \text{ trials}) &\leq (1-p)^t \\ &\leq (e^{-p})^t \quad \left. \vphantom{\Pr(\text{failure after } t \text{ trials})} \right\} 1+x \leq e^x \\ &= \frac{1}{e^{pt}} \end{aligned}$$

IDEA: Use  $t = \lceil 1/p \rceil$  to achieve "constant error probability".

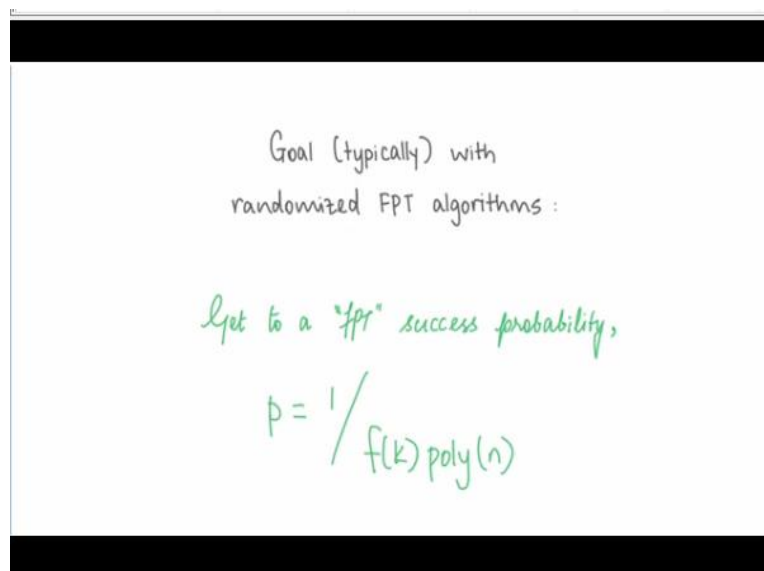
Now let us just simplify this a little bit further. So, because of this inequality  $1 + x$  at most  $e$  to the  $x$  we can say that  $1 - P$  to the  $t$  is at most  $e$  to the  $- P t$ . So, just writing it out as a fraction hopefully it is visible that essentially if you do  $1$  over  $P$  repetitions of your experiment then you have a constant bound on the error probability. By a constant bound I



just mean that there is no dependence on the input size it is a universal constant in this case something to do with  $1/e$ .

Now if you want to boost your success probability even further you could run your algorithm say 100 times  $1/P$  if you like and in practice I guess there is a little bit of experimentation involved in trying to figure out exactly how many times you want to run the algorithm before you can be reasonably confident. But I can assure you that if you actually run the numbers then with a fairly reasonable number of repetitions you can get error probabilities that are so small that they are practically negligible and you do not have to worry about them.

**(Refer Slide Time: 15:42)**



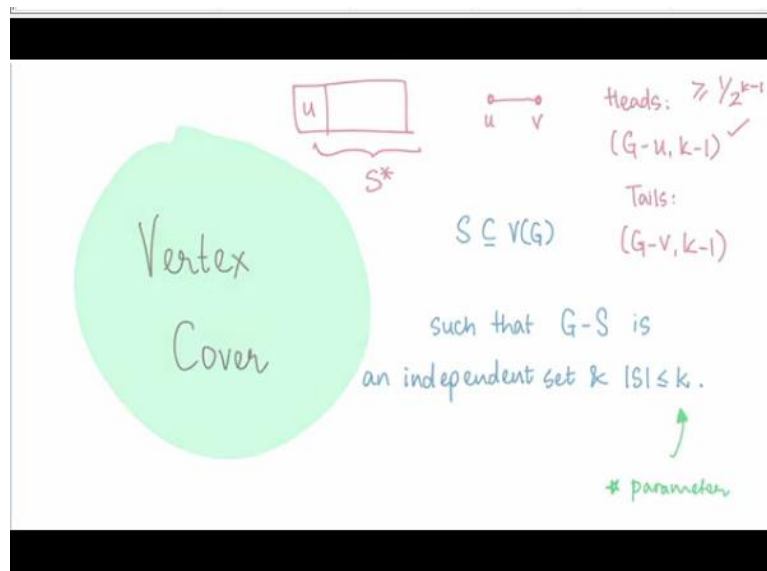
Now what all this means in the FPT context is that we will typically be trying to design algorithms which have a FPT formatted success probability. So, we will usually end up designing polynomial time algorithms which have one-sided error with false negatives where the success probability is at least  $1/f(k) \text{ poly}(n)$  for some  $f(k)$ . What this will ensure is that there is an algorithm which runs in FPT time which is because we are going to be boosting the base algorithm by running it  $f(k) \text{ poly}(n)$  many times.

That is the repetition strategy that we just discussed and if we run it so many times then we get an algorithm that has a overall constant error probability and that is just going to be our gold standard in the context of this discussion. So, if we get there then we are going to take that as pretty much a job well done and these are the kind of algorithms that we will be designing throughout this week.

So, we will not come back to the same argument multiple times, so what we will basically do is talk about a problem and come up with the base algorithm which runs in polynomial diamond which has this kind of success probability and because of this discussion that we have just had that that also implies automatically an algorithm with a FPT running time that has a constant error probability bound.

So, with all this background in place we are finally ready to design our very first randomized FPT algorithm. The problem that we are going to design this algorithm for is a problem that you are by now very familiar with and no points for guessing, it is vertex cover.

**(Refer Slide Time: 17:28)**



If you have noticed that this lecture was supposed to be about feedback vertex set, you might be a little bit worried by vertex cover showing up here, but do not worry. We really are going to be talking about feedback vertex set in just a minute, but it is a FPT tradition to start with vertex cover whenever possible and it turns out that vertex cover really does have a really nice randomized FPT algorithm and it makes for a perfect warm-up exercise.

So, I will not actually be describing the algorithm in complete detail but I will give you a couple of hints, so that you can get started and I would really recommend pausing here and trying to work out the details for yourself before progressing to FVS. So, for vertex cover a useful thing to remember is that whenever you have an edge say between vertices  $u$  and  $v$  any vertex cover must pick one of these 2 endpoints.

So, here is a natural randomized strategy. So, if your graph  $G$  is empty to begin with then there is no work to be done. But if it has at least one edge then just pick your favorite edge it does not matter which one; you know that at least one of these endpoints are going to be useful to you. But you do not know which one, but in a randomized algorithm whenever you are unsure you can toss a coin.

So, you have an edge on your hands and you do not know which end point to pick, so let us toss a coin and let us say that if the coin toss turned out heads then we pick  $u$  which is to say that we recursively get into the instance  $G - u, k - 1$  and let us say that we had tails instead then we look at  $G - v, k - 1$ . So, this might remind you of the branching algorithm that we discussed in the second week.

These recursively generated instances do look rather familiar. But remember that unlike the branching algorithm we are not exploring both of these possibilities simultaneously we are getting into just one of them based on a coin toss and the point is that you could imagine that these random bits that you are drawing are basically dragging you through the search tree that we had developed back then.

They are taking you down a very particular path and if you were lucky then it would be the right path and how lucky can you hope to get? Well at every step you make a right choice with probability at least half. So, if all of these right choices sustain across levels then you have made it and you have made it with probability at least  $1/2^k$  because this recursion only goes down  $k$  levels deep.

Let me just briefly hint at how you might formalize this line of reasoning. So, suppose you were working with a YES instance that means that there is some vertex cover of size at most  $k$  in this graph  $G$  that you are working with, so let us fix some specific vertex cover of size at most  $k$  and let us call that  $S$  star. Now we know by virtue of being a vertex cover  $S$  star must contain at least one of  $u$  or  $v$ , it possibly contains both but it contains at least one.

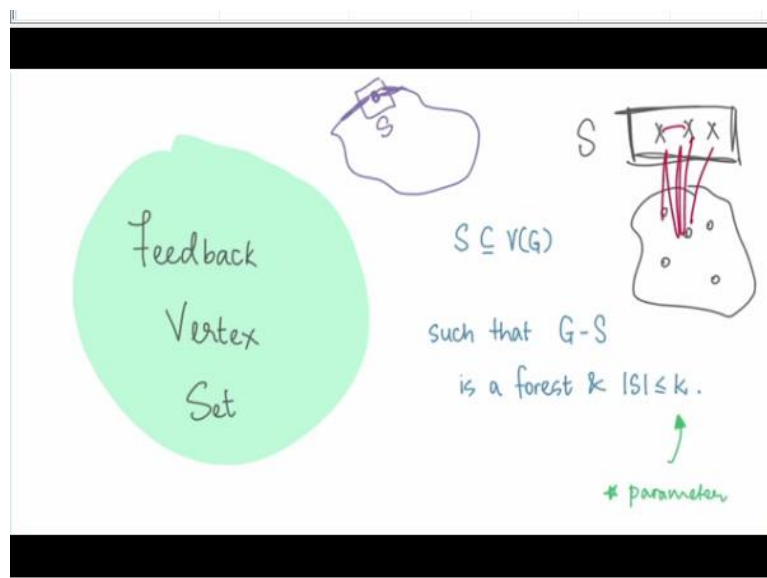
And we also know that with probability one half we get into the correct recursive call. So, in this case the right call to get into would be this one. Notice that  $S$  star -  $u$  is a witness for the fact that  $G - u, k - 1$  is a YES instance. So, let us say that the recursive call invokes a run

of the algorithm which works out with a success probability of at least  $1/2^{k-1}$ . This is our inductive hypothesis.

So,  $G - u_{k-1}$  will say YES with probability at least  $1/2^{k-1}$ . It might be because it is discovered a vertex covered different from  $S^* - u$  possibly but that is not relevant here. The point is that it gives you the correct output with probability at least half to the  $k-1$  and combined with the probability of getting into this invocation in the first place which is one half, you get an overall success probability of half to the  $k$ .

Now remember that when you are convincing yourself about why everything works out make sure to fill in the little details like what the base cases are, what the possible terminations are, try to observe what happens when you start off with the NO instance and things like that. Once you have played around with this a bit and you are comfortable with what is happening then make sure to continue, join us back to talk about feedback vertex set.

**(Refer Slide Time: 22:32)**



So, let us begin by pulling up the definition for a feedback vertex set. By now this should be a pretty familiar problem, we saw it for the first time when we were talking about branching algorithms and if you remember back then what we did was apply some reduction rules after which we said we can comfortably branch on the top so many high degree vertices and that was the main idea there that led to a running time of the form  $k$  to the  $k$ .

Then we saw it when we discussed iterative compression and there we improved this running time to  $5$  to the  $k$  and today we are going to improve the running time further to  $4$  to the  $k$  of

course with the caveat that we are working in the randomized setting. I would not worry about this too much in practice because as we just discussed we can get the error probabilities down to a point where they are negligible in practice.

But of course theoretically you would want to make a distinction between the best running time that you can achieve with randomization and the best running time that you have among deterministic algorithms. So, that it is always an apples to apples comparison. So, now that all that advertising is done and hopefully we have built up some anticipation for the algorithm. Let us actually talk about the algorithm now.

So, remember that a feedback vertex set is trying to destroy all the cycles in the graph. So, you might want to take a moment to think about what would be a natural randomized strategy for coming up with an optimal FVS if it exists. Remember that we are designing these randomized algorithms with one-sided error. So, we are mostly concerned about the YES instances; we want to make sure that if the graph did have a feedback vertex set of size at most  $k$  that we get to it with a reasonable probability.

So, we are going to be picking vertices at random and we are going to hope that these vertices end up being inside some arbitrary but fixed FVS of size at most  $k$ . So, to realize this hope it is often useful to visualize what a typical YES instance looks like. So, let  $S$  be some arbitrary but fixed FVS of size at most  $k$  and we know that the rest of the graph is of course a forest.

And we want to hit the lottery by somehow being able to get to the vertices in  $S$  with whatever randomized experiment we come up with. So, that is clearly the goal, but how do we actually come up with such an experiment? Well let us think back to vertex cover where we actually managed to do this right. So, what helped us in vertex cover was the structure of  $G - S$ . So, what was happening in  $G - S$  was that there were no edges at all.

So, every edge in the graph was either completely inside  $S$  or was going across  $S$  and  $G - S$ . So, if we pick any edge in the graph then with at least probability half we would actually get to a vertex in  $S$ . More generally you could say that if a good fraction of all of your edges are going across between  $S$  and  $G - S$  then if you do the same thing which is that you pick an edge at random and then you pick one of its end points at random then there is a good chance or some chance that you will end up inside  $S$ .

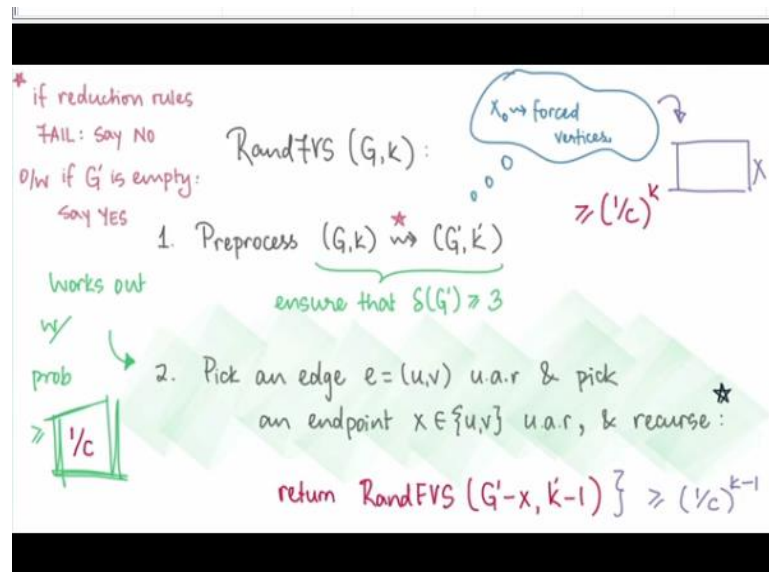
Now this some chance will be quantified by the fraction of edges that go across relative to the total number of edges. So, hopefully this is clear if you need a moment to pause and absorb this please do because this is the main intuition that will drive our algorithm. But if you are observant about the structure of FVS you might already start pointing out that this strategy may not work very well.

Because for example what if your graph was just one long cycle then there is a FVS that just picks 1 vertex, it has 2 edges going out of it and clearly this is not a decent proportion of the total number of edges in the graph, it is certainly not a constant fraction and it is a fraction that depends on  $n$  which means that we are not going to get the anticipated FPT running time for such instances.

So, remember that we did have some reduction rules that were quite useful when we did the branching algorithm, in fact we also used reduction rules for iterative compression and it turns out here also it is these reduction rules that will come to our rescue. So, notice that even in this example of one straight long cycle, one of the issues is that you have a lot of these degree 2 vertices.

But remember that we do know how to pre-process degree 2 vertices. Of course you might say okay look this is just one example and you took care of it like this, but in general why should I expect that a lot of edges will be crossing  $S$  and  $G - S$  for a generic reduced instance. So, that is exactly the thing that we will try to formalize but first let me tell you the algorithm. So, you know what is it that we are going to need for this algorithm to work.

**(Refer Slide Time: 27:58)**



So, as I said we are going to begin by doing some preprocessing. For now this is just because we can so why not it does not hurt you get it for free. But we will see that this preprocessing is crucial and really useful for getting to the probability bounds that we desire later on. So, just to be specific these are the same reduction rules that we had when we were discussing the branching algorithm for FVS

So, let us just quickly review the 5 reduction rules that we introduced back then. In the first one we said that if there is a self loop at a vertex  $V$  then we will essentially remove  $V$  and reduce the budget by 1. In the second reduction rule we said that if you have more than 2 edges between a pair of vertices then we reduce the multiplicity to exactly 2. The third reduction rule said that if you have a vertex whose degree is at most 1 then you delete this vertex and the budget remains unchanged.

So, this applies to vertices of degree 1 and also to vertices that are isolated. So, you remove this vertex but leave the budget as it is. In the fourth reduction rule we discussed bypassing or short-circuiting vertices whose degree was exactly 2 and once again in this case what we did was we added an edge between the neighbours of this vertex and we also left the budget unchanged.

And finally in the last reduction rule we said that if we run out of budget completely then we can simply say no. Another way of writing this would be to say that if  $k$  hits 0 and  $G$  is not already a forest then you can say no. These 2 ways of articulating this final reduction rules are pretty much equivalent. So, I leave it here these are the 5 reduction rules if you need some

time to recap them and remind yourself why all of this works as you expect then please feel free to take a pause here.

Now what these reduction rules do ensure as we have discussed before a few times is that when they have been applied exhaustively and you have not already resolved the instance then we know that we have a non-trivial graph where the minimum degree is at most 3. So, in the next step what we are going to do is to say that if these reduction rules fail and they declare that you have a NO instance then of course  $r$  algorithm can also stop and take advantage of this information.

And also if these reduction rules stop and you get to a point where after these reduction rules have done their job the graph becomes empty then you say yes. So, for instance this would happen if your graph was just a cycle to begin with the example we were discussing earlier because what is going to happen is that you will keep applying the short circuiting rule to a point where you get to a cycle which is of length 2.

At that point it is going to collapse and become a cycle of length 1, at that point if you had a budget of at least 1 then at that point you are just going to include that vertex and your graph becomes empty and in this situation  $r$  algorithm will also happily conclude that we were dealing with a YES instance and it is going to stop at this point. But on the other hand if you are starting budget was 0 then of course the last reduction rule will come into action.

And it will inform you that on an input graph that is a cycle you cannot have a FVS with 0 vertices. So, one way or the other this sort of a graph will be completely handled by the reduction rules themselves. Now for  $r$  algorithm what we are going to do is just keep track of the vertices that were forced by the reduction rule and let us say that we have not said YES or NO at the end of step 1.

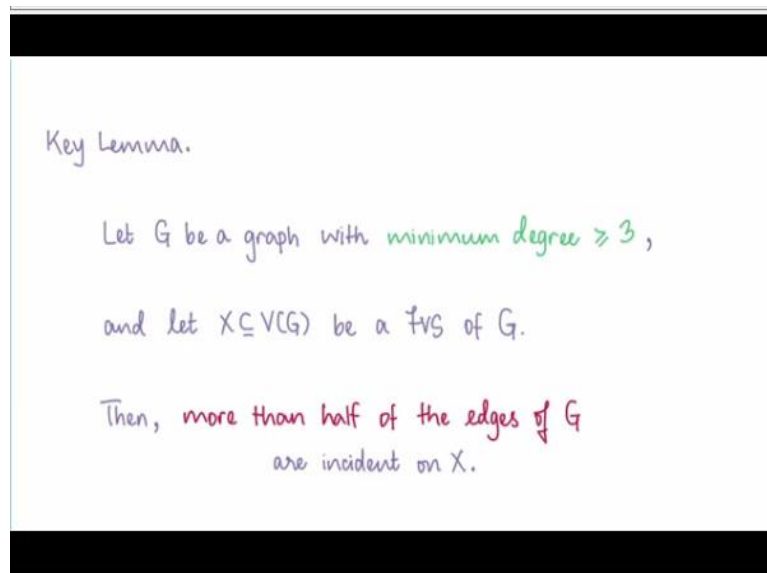
That means that we are left with a graph that is non-trivial in the sense that it has at least one edge and it has a non-trivial budget and at this point we can move on to the next phase of the algorithm where we actually perform the random experiment. So, hopefully with all this pre-processing in place we get to a situation where we can have a decent congestion of edges that go across any hypothetical solution.



And the forest that is left behind if this was a YES instance. So, the random experiment itself is very simple all that we do is we pick an edge uniformly at random and we pick one of its endpoints uniformly at random and this is the end point that we want to include in our solution. So, we are going to recursively invoke this randomized algorithm on  $G - V$  with a reduced budget of  $k - 1$ .

So, this is our hope that this vertex rather that we have picked with this random experiment is actually sitting in some arbitrary but fixed solution if  $G, k$  was a YES instance. So, notice that there are really exactly 2 places where we do something at random. The first is when we pick the edge and the other is when we pick one of its end points. And let us now try to formalize the intuition that we have been discussing all along.

**(Refer Slide Time: 33:26)**



So, here is the statement that we would like to prove. If a graph has minimum degree at least 3 then any feedback vertex set of this graph must actually have a lot of edges from the graph incident on it and in particular the way we quantify this is to claim that strictly more than half the edges of the graph must have at least one of their endpoints in the FVSX. So, with this specific statement let us go back to the algorithm and analyze the probability of the good event.

Remember that the good event for us is that when we pick an edge at random and pick one of its end points at random what is the probability with which we choose a vertex in some fixed FVS. Remember that are working assumption is that we are dealing with a YES instance, so we can fix FVS of size at most  $k$  at the back of our minds and work with that. Based on the

previous statement which was that any such FVS must in some sense absorb at least half of the edges that are there in the graph in fact strictly more than half what can you say about the probability with which we get lucky in the random experiment that we perform.

Notice that the answer to this question here really drives the entire analysis about the probability with which your overall algorithm succeeds which in turn determines the running time of your randomized FPT algorithm. So, in particular suppose you are able to conclude that things work out with probability at least  $1/C$  then notice that you can actually establish a probability of  $1/C^k$  for the success of the overall algorithm.

And the reason you can do this is very similar to the inductive approach that we had when we were working with vertex cover. So, based on the induction hypothesis you can conclude that this recursive call will do the right thing with probability at least  $1/C^{k-1}$ . And the entire algorithm works out if the recursive call works out and if your top level decision was the right one and the combined probability for that is going to be at least  $1/C^k$  because it is just the product of these 2 terms here.

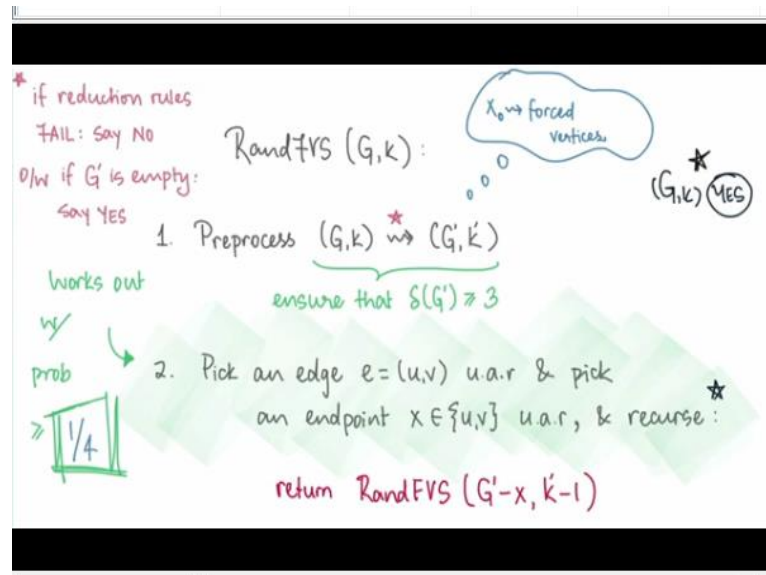
Now there are some details that do need to be worked out and specifically the detail that I am skipping here is the one about how the pre-processing step interleaves with the randomized step. But the intuition here is that the pre-processing step basically cannot hurt because you have a fixed FVS in the back of your mind with respect to which you are working out these probabilities.

And notice that these vertices  $x_0$  actually belong to this FVS for sure because by design our reduction rules only force those vertices which belong to any FVS. So, for these vertices in fact we can be quite sure that we are never making a mistake anyway. So, in some sense the idea is that the vertices that are chosen by the reduction rules they never hurt and for the vertices that you are choosing with the help of the random experiment you get the right thing with a decent enough probability that you can say something meaningful about the success of the algorithm overall.

So, do make sure that you work through the details of how these pieces fall together to give you the whole argument but for now let us try to concentrate on what this constant  $C$  should be. So, that we know that we have an algorithm whose running time is  $C^k$  based on the

boosting arguments that we had set up earlier. So, once again keeping in mind that you promised that more than half the edges of the graph  $G$  have at least one of their endpoints in  $x$  what can you say about this constant  $C$ . This is a perfect time to pause and think about this and come back when you are ready let me just clean up the slide in the meantime.

(Refer Slide Time: 37:34)

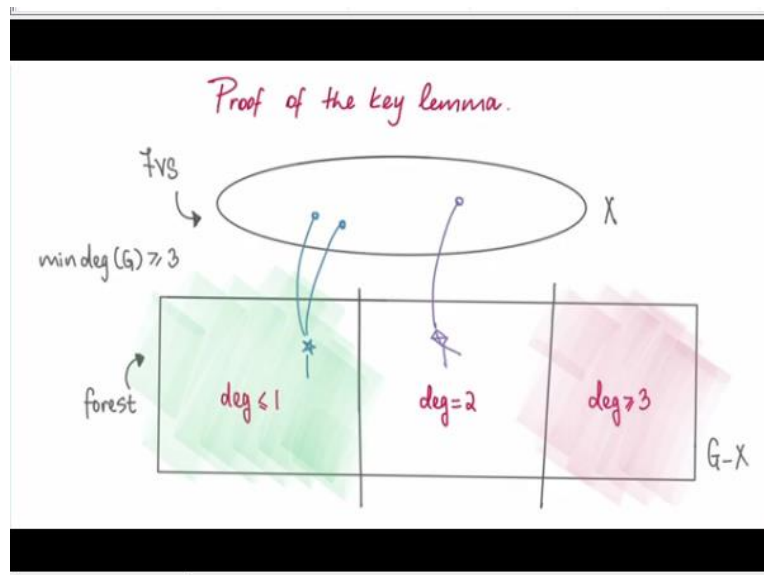


So, if you are ready for it the answer turns out to be one-fourth and the reason for this is that each of the 2 experiments that we are performing individually work out with probability at least one half. The first step where we are picking an edge at random in fact works out with probability strictly more than one half based on the lemma that we stated before. Of course we are yet to prove the lemma but if you believe the statement then this step gives you an a good edge with probability more than half.

Now having landed on a good edge which is an edge with at least one of its endpoints in  $x$  we pick the correct endpoint the one that sits in  $x$  with probability at least half. Of course if you are really lucky it may be an edge which has both of its endpoints in  $x$  in which case this step will actually work out with probability 1, but even if you are unlucky in the worst case you will still get to a vertex which sits in  $x$  with probability at least half.

So, the overall probability is certainly at least one fourth and this is what gives you the 4 to the  $k$  algorithm modulo the correctness of the key lemma. So, of course hopefully we are now fully motivated to actually prove this key lemma here and that will truly complete the description of this algorithm. So, let us get started on this.

(Refer Slide Time: 38:53)



To begin with let us depict the FVSX and the forest  $G - X$  with this picture and the first thing we are going to do is classify the vertices of the forest into 3 categories. So, these are the vertices that have degree at most 1 degree exactly 2 and degree at least 3. Now of course you might complain a bit because you might say it was not this a graph which has minimum degree at least 3. So, what do you mean by vertices that have degree at most 1 or degree 2?

Well I am going to be talking about the degree in  $G - X$ . Notice that  $G - x$  is a forest, so when I talk about the degree here it is the degree that these vertices have only inside  $G - X$  and in fact that brings me to my next point because  $G$  overall has minimum degree at least 3 we know that these vertices must have their deficit degree many neighbours in  $x$  because that is the only other place that they can find these neighbours.

This is what we need to ensure that it is true that  $G$  has minimum degree at least 3. So, all the leaf vertices and isolated vertices of the forest  $G - X$  must have at least 2 and at least 3 more neighbours respectively in the FVSX and every vertex that has degree exactly 2 in  $G - X$  must have at least 1 neighbour in  $x$ . If this is just based on the assumption that  $G$  has minimum degree at least 3.

So, notice that this picture is already looking pretty promising because we are able to generate some edges that are going across and the number of these edges is actually **of** some subset of vertices in  $G - x$ . That is all that we have for now, but let me also say that we will actually be proving this lower bound on just the number of cross edges, we will not even

worry about edges with both of their endpoints in  $x$  will say that even just the number of cross edges happen to be more than the number of edges that sit inside the forest.

And in fact instead of working with the number of edges in the forest let us work with the number of vertices in the forest because that is just going to be slightly more convenient and it is again good enough because in fact the number of vertices in a forest is anyway more than the number of edges. Remember that in a forest the number of edges is at most  $n - 1$ . So, if we show that the number of cross edges is more than the number of vertices in the forest then anyway we have that the number of cross edges is more than the number of edges in the forest.

So, if you do not like thinking about things in terms of pictures we will anyway write this down again in terms of inequalities in just a couple of minutes. But I am saying all this here right now just so that you have some intuition for what is it that we are trying to do. So, just to recap what we will try to show is that the number of edges that go across is strictly more than the number of vertices in the forest.

And notice that this will actually imply the claim that we want to prove. So, we already have that the edges that are incident on the degree 2 vertices are in 1 on 1 correspondence in particular every degree 2 vertex is any way sending one edge to  $x$ , so there we are pretty much done in terms of what we want to show. The vertices that have degree at most 1 are actually sending at least 2 edges.

But the vertices that have degree at least 3 are sending no edges. So, this seems like we are in a very good situation with respect to vertices of degree at most 1, but we are in a bad situation with respect to vertices of degree at least 3. But remember that in a forest we also have a relationship between vertices that have degree at least 3 and vertices that have degree at most 1. Intuitively vertices of degree at least 3 can be thought of as branching vertices.

And the more branching vertices you have the more leaves you end up spawning because of them. So, formally you can prove this by induction but it turns out that the number of degree at most 1 vertices is strictly more than the number of degree at least 3 vertices in any forest. So, with that we know that because these vertices that are highlighted in green are anyway sending at least 2 edges.

In some sense they take the responsibility of accounting for the vertices of degree at least 3 as well. So, hopefully this establishes the intuition for everything that we are trying to do and now let us just go over the same argument one more time. This is going to be a recap, but in notation and with inequalities written out.

(Refer Slide Time: 43:37)

Want to show:  $\# \text{ edges inc. on } X > \frac{m}{2}$

Equivalently:  $\# \text{ edges inc. on } X > \# \text{ edges not inc. on } X$

Suffices to show:  $\# \text{ crossing edges} > \# \text{ edges in } G-X$

Also enough to show:  $\# \text{ crossing edges} > \# \text{ vertices in } G-X$   
 $\geq \# \text{ edges } (G-X) + 1$

So, remember what we are trying to show is that the number of edges that are incident on a feedback vertex at  $x$  is strictly more than  $m$  by 2 and keep at the back of your mind that we have a graph where the minimum degree is at least 3. So, this is the same as trying to show that the number of edges incident on  $x$  is strictly more than the number of edges which are not incident on  $x$ . That will say that these edges account for more than half of what is available.

Now the first thing we said is that it is enough to just focus on the crossing edges. So, we will in fact show that even if you just had the crossing edges they alone account for more than half of the total number of edges and in particular we will show that the number of crossing edges is more than the number of edges in the forest and the next thing we said is that instead of worrying about the number of edges in the forest let us work with the number of vertices in the forest because again that is enough and that is just more convenient.

So, finally the thing that we will show is that the number of edges that cross  $x$  and  $G - X$  which is to say they have 1 endpoint in  $x$  and 1 endpoint in  $G - x$  the number of such edges is

strictly more than the number of edges which have both of their endpoints in  $G - X$  which is the forest.

(Refer Slide Time: 44:52)

The slide contains a handwritten diagram and a mathematical derivation. The diagram shows a rectangle labeled  $X$  at the top and an oval labeled  $G-X$  at the bottom. The oval is divided into three vertical sections labeled  $V_1$ ,  $V_2$ , and  $V_3$ . Three blue dots are shown: one in  $V_1$ , one in  $V_2$ , and one in  $V_3$ . Blue lines connect these dots to the rectangle  $X$ , representing crossing edges. The text  $\# \text{ crossing edges} > \# \text{ vertices in } G-X$  is circled in red at the top. Below it, the word **(GOAL)** is written in red. To the right of the diagram, the following derivation is written:

$$\begin{aligned}
 \# \text{ of crossing edges} &\geq 2|V_1| + |V_2| \\
 &= |V_1| + |V_1| + |V_2| \\
 &> |V_1| + |V_3| + |V_2| \\
 &= |V(G-X)|
 \end{aligned}$$

So, let us just work through this now, remember we had this partition of the forest into 3 classes which here I will denote by  $V_1$ ,  $V_2$ ,  $V_3$  for convenience. So,  $V_1$  is the leaves and the isolated vertices when we focus on the graph induced on  $G - x$ ,  $V_2$  is all the degree 2 vertices of the forest and  $V_3$  is all the vertices that have degree at least 3 again in the forest. So, the first thing is that we have the number of crossing edges being lower bounded by 2 times  $V_1 + V_2$  this is just based on the fact that  $G$  has minimum degree at least 3.

So, these guys must have respectively at least 2 and at least 1 neighbour in  $x$ . The next thing is that you can just write out 2 times  $V_1$  as  $V_1 + V_1$  and the reason we are splitting it this way is so that we can substitute one of these  $V_1$ 's with  $V_3$  and remember we said that the number of vertices that have degree at least 3 in the forest is strictly more than the number of vertices that have degree at most 1.

So, with this written out this is going to be very clear now because  $V_1 + V_2 = V_3$  I mean when you account for the sum of their sizes you get exactly the number of vertices in the forest  $G - X$  and that is exactly what we wanted to show. So, with this we have the proof of the key lemma and based on the key lemma we have a 4 to the  $k$  randomized algorithm for finding a FVS of size at most  $k$ .

Again remember that the algorithm itself was really simple we just applied the 5 reduction rules that we already had when we were discussing the branching algorithm. So, you just get rid of vertices with loops on them; you reduce the multiplicity of high multiplicity edges down to 2, you short circuit the degree 2 vertices, you eliminate or delete vertices that have degree at most 1 and you say no whenever you run out of budget.

So, these were the pre-processing rules that ensured that we either were able to completely resolve the instance or we ended up with a graph where the minimum degree was at least 3 but this minimum degree at least 3 gave us enough structure to ensure that a random experiment which was simply pick an edge at random and pick one of its end points at random worked out and gave us a vertex from the FVS whenever it exists with probability at least one fourth.

And this is what gives us the  $4$  to the  $k$  randomized algorithm. So, with this we come to the end of this lecture, thanks so much for watching. I hope you enjoyed this and as usual we will see you either on the mailing list or on the discord channel if you have any questions about this at all and in the meantime I will see you in the next module.