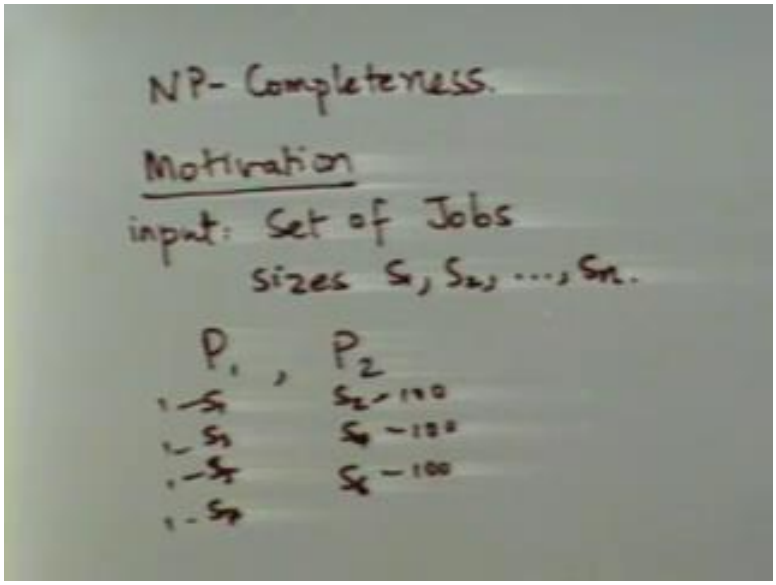
Design and Analysis of Algorithms
Prof. Sunder Vishwanathan
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 26
NP - Completeness - I
Motivation
Introduction to Reductions

(Refer Slide Time: 00:51)



Deals with the notion of NP completeness, we will study this notion and we will see how to use this notion effectively. The first thing I will sort of talk about is motivation. Hopefully at the end of this I will motivate you sufficiently that, you will be interested enough to study about the subject. So, imagine that you finished your B Tech. You have the all new subjects well, and then you gone for a interview, you land your dream job. The company pays well, you are really happy, you settled in software job.

You very happy, because it is you have to deal with designing things. And your boss is also happy with you and he wants to give you challenging projects. The first project you get is the following it scheduling jobs on computers. So, the input is set of jobs and each job as a size associated with it. So, let us say the jobs are j 1, j 2 up to j n and the sizes are s 1, s 2 and so on to s n.

These sizes could vary, they could some of the jobs could have the same size, some could have different. You can think of them as positive integers. Now, there are two processors. So, there are two processors p 1 and p 2 both are identical. And your job is to schedule, these jobs on these two processors.

So, you want to automate your boss wants you to automate this process. You have to write a program, which takes input these sizes s 1, s 2 up to s n. So, let us say a array of size n or reads it of a screen. And you have to schedule these jobs on processors p 1 and p 2. Now, the scheduling should be such, that the last job finished is fastest. So, what do we mean by this?

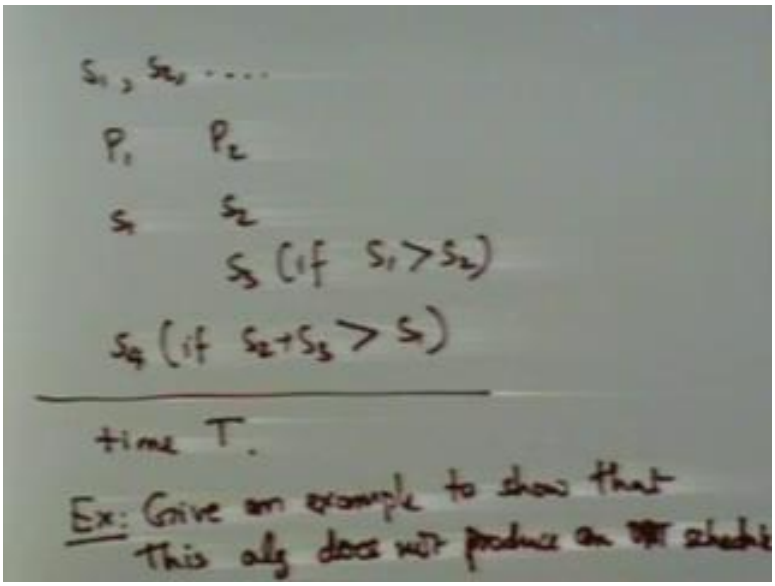
Supposing, I schedule all of them in p 1. And the time taken will be s 1 plus s 2 plus s 3 and so on up to s n. Let us say that job of size s i takes the same time, time s i to finish if I schedule it, it takes time s i. So, also assume that there is preemption. Once, you start a job of it has to run for completion. So, if I run all of them on one processor, the time taken is s 1 plus s 2 plus s 3 up to s n.

So, at time s 1 plus s 2 up to s 3 and so on up to s n, all jobs must finish. Now, supposing I schedule, the even jobs on p 2 and odd jobs on p 1. Let us say s 1, s 3 and so on here it is s 2, s 4. Let us say there are seven jobs s 5, s 7, s 6. So, this is how the jobs are scheduled. Now, the time taken on p 1 is s 1 plus s 3 plus s 5 plus s 7, time on p 2 is s 2 plus s 4 plus s 6.

Now, it may so happen that these are very large jobs. Let us say, these sizes are 1 and these are 100. Then, the time at which the last job completes is 300. This process finishes of in four units of time, while this takes 300 units of time. So, the last job finishes after 300 units of time, this is not what we want. So, you want to sort of distribute the jobs evenly among these two processors.

So, you want to distribute the jobs in such a way, that these large jobs say one of the s 4 should be here. And may be s 5 should be here and so on, we want to distribute as evenly as possible. So, you think about this for some time and here is the first idea, we come up with.

(Refer Slide Time: 06:07)



So, you take the jobs in order. So, the jobs are of sizes s 1, s 2 and so on with the first job on the first process it is p 1 I put s 1 here. The second job goes to p 2. Now, you put when I take the third one, you put it on the processor, which is least lightly loaded. For instance, if s 1 is greater than s 2, then I put s 3 here. And you keep doing this, I look at s 4 and I see which one is greater.

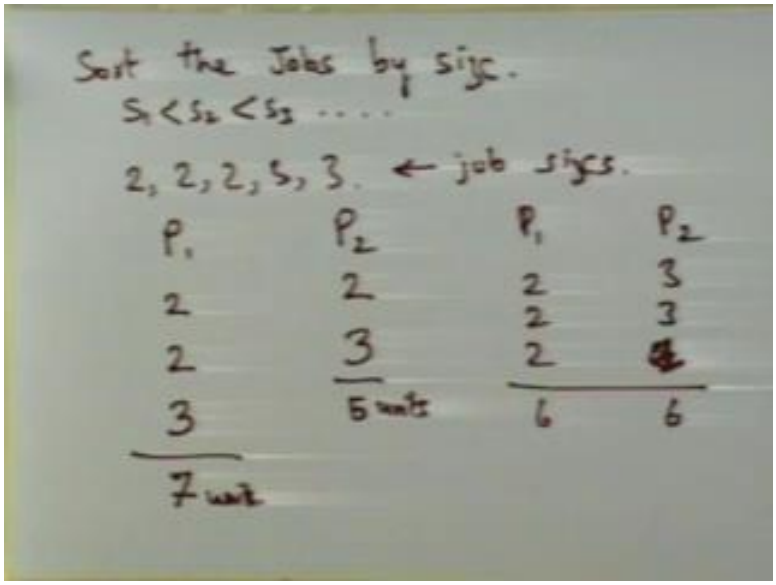
Assume that s 2 and s 3 are on p 2, check whether s 1 is larger than s 2 plus s 3 or is s 2 plus s 3 larger than s 1. If s 2 plus s 3 is larger than s 1 then I put s 4 on p 1. So, s 4 will go here, if s 2 plus s 3 is larger than s 1, you can break ties as you want. This is the first algorithm you come up with. You code this and it is starts working your boss is happy. But, the next day the boss shows up and tells you, that this does not work.

So, he gives you an input on which you produce the schedule, which takes some time T. So, you take some time T while the boss produces the schedule which is less than T. So, the optimal schedule on the input is smaller, the total time the optimal takes is smaller than the time that your algorithm produced. And the boss says fix this. So, at this point I would like to give you an exercise, what give an example to show that, this algorithm does not produce a optimal schedule.

So, give an example to show that this algorithm does not produce the optimal schedule. Well, you go back and figure out, there is something wrong with the algorithm in the

sense, that if these jobs are mixed up. If the sizes are mixed up arbitrarily, may be that is the reason the algorithm does not work.

(Refer Slide Time 09:10)



So, here is your next attempt, the first thing you do is you sort the jobs by size. Now, you know that s 1 is less than s 2 less than s 3 and so on. You know that, the jobs are increasing order of size. And now you do the same algorithm, you put s 1 on the first processor, s 2 on the second and so on. Well, this seems to work, but two days later again your boss shows up with an example, which does not work.

So, let us look at this example. The example has five jobs and the sizes are as follows 2, 2, 2, 3 and 3, let us see what your algorithm does. So, here are my two processes p 1 and p 2, these are the job sizes. So, the first job goes on to p 1, the second job is on p 2, the third job you could put either on p 1 or p 2, let us say we pick p 1. The next one goes here and well 4 is smaller than 5. So, you put the last one here.

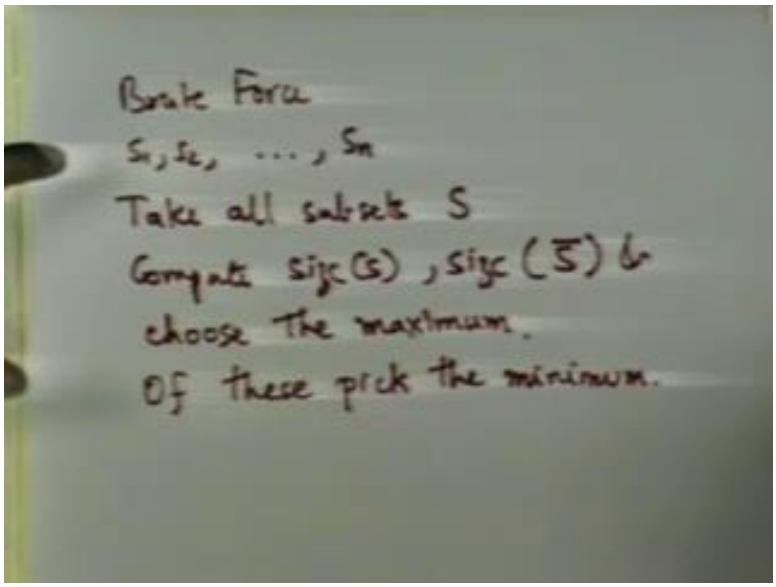
Now, the total time that your schedule takes is 3 plus 2 plus 2 which is 7. On this processor of course, you take 5 units on this you take 7 units. So, the total time to finish all jobs is 7 units. Well, you can see that this is not optimum, the best way to do this is I guess you have seen them by now, you put the 2s on the one processor and the 3s on the other. Now, there are only 2 3s. So, this takes 6 units and that takes 6 units every job has finished in 6 units of time.

So, this ((Refer Time: 11:19)) that you will come up with also does not work. Let me again repeat the algorithm, you sort the jobs by size. Now, you look at these jobs one by one. And assign these processes greedily, in the sense that, you put the first one in the first process, the second one in the second. And when I reach the ith job, you put it on the least lightly loaded processor.

You sum up the previous sizes, which are on each processor, choose the smaller one and put this new job on that processor. This was the algorithm we came up with and this does not work, here is an example that it does not work. So, now what do you do? Well, you can go back and try fix this example. When, you come up with another heuristic, which beats this example.

But, I can tell you that it is extremely difficult possibly impossible to come up with a smart heuristic like this, which does well for all inputs, which always does well. So, you after trying various heuristics most of them do not work. You are now at the end of your patience, you are really worried about what your job, because boss is now threatening you with dismissal. So, then what you do is this. So, you now sort of want an algorithm that works always. So, what you do is a brute force algorithm.

(Refer Slide Time 13:11)



So, you take every subset of the jobs. So, I have jobs, I have sizes s 1, s 2 up to s n. So, take all subsets ((Refer Time: 13:30)). So, compute the size of s and size of S bar. S bar is the complement of S. So, all jobs not is S which is S bar. So, essentially we want to

put s on process on p 1 and the complement on the processor p 2. So, you do this and look at the maximum of these two and choose the maximum of these two.

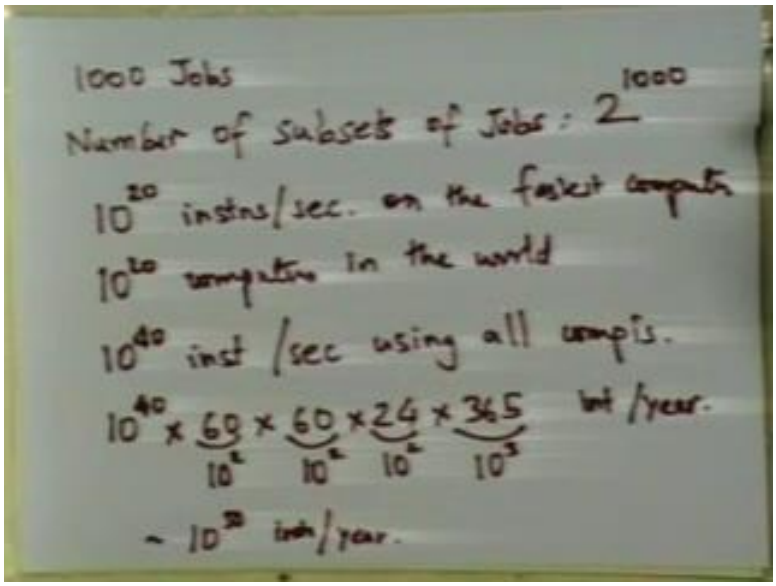
So, this will tell you the time, the total time that this schedule took. So, do this for every possible subset and pick the minimum. Take all subsets compute these choose the maximum and of these maximum pick the minimum of these. You can easily see that this will always give you the right answer. Because, you will look at all possible ways of scheduling, these jobs on two processors and one of them has to be the minimum.

Now, this is fine, this algorithm works, your boss tries out a few inputs and it is all it works for all most all for... In fact, for all inputs till your boss feeds in an input with let us say 1000 numbers. There are 1000 jobs, which have to be read to these processors these two processors. So, we give the input as a list of size 1000.

Now, you start running the algorithm 5 minutes pass an hour passes, 2 hours, 10 hours, 1 day the algorithm does not stop, 2 days, one week. And now your boss is starting to get worried. Here is an algorithm which is got for all small inputs and now it is weak and the algorithm does not stop for an input of size 1000. Well I can tell you that this is not going to stop even after many years.

You can take years and years and this is not going to stop, in my lifetime not your lifetime may be many lifetimes. That is a problem with such brute force approaches. Let us do a quick calculation to see how much time your algorithm is going to take on inputs of size 1000.

(Refer Slide Time 16:30)



If you have 1000 jobs, the number of subsets of jobs that you look at is to be 1000. So, the number of subsets of jobs is 2 to the 1000. Now, let us assume that in one instruction a computer can process one of these subsets. Let us see, how much time computers will take to process all of these subsets. Now, the fastest computer runs, let me make an estimate. Let us say 10 to the 20 instructions per second.

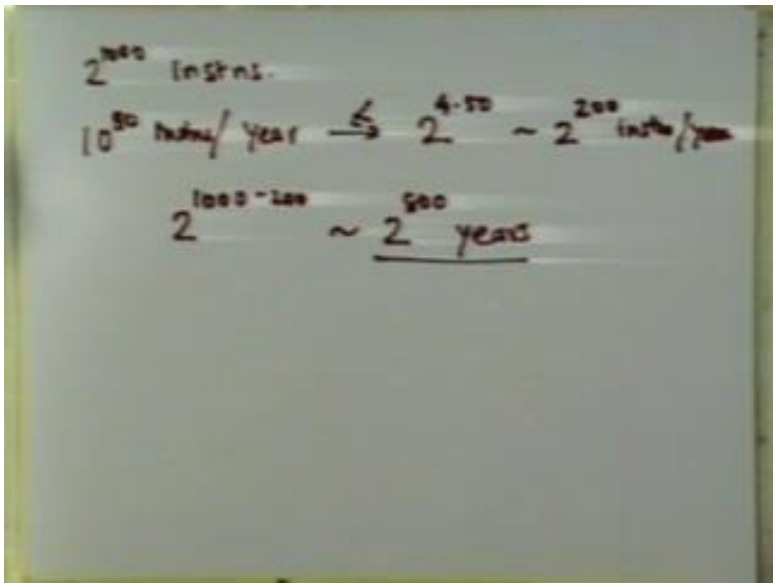
This is the fastest computer. On the fastest, this is a big over estimate it is much smaller than this, but we will assume this, how many computers you think there are in world 10 to the 20. Let us say 10 to the 20, it is again an overestimate computers in the world. What we are going to do is put all the computers in the world, at your disposal to solve you know to run the your brute force algorithm.

So, putting these two together, then I have 10 to the 40 instructions I can do instructions per second, using all computers. So, I am using all the computers in the world and I can do 10 to the 40 instructions per second. So, how many instructions do you think, one can do in a year, let us do this calculation. So, it is 10 to the 40 instructions per second. So, in an hour I can do this time 60 in a minute I can do this time 60 another 60 for an hour. Then, I have 24 hours a day.

Then, I have 365 days in a year. So, these many instructions per year. Let me do a overestimate here, this I am going to say 10 to the 2, this let us say this is 10 to the 2. Well, 10 to the 2 since we are generous, this is 10 to the 3. So, number of instructions per

year is certainly not more than 2, 4, 6, 10, 10 to the 50 good. So, I have 2 to the 1000 subsets to look at. I can look at at most 10 to the 50 subsets per year. Let us see how many years this takes. So, let me quickly.

(Refer Slide Time: 19:53)



So, you have 2 to the 1000 instructions and using all the computers in the world the fastest speeds. You have 10 to the 50 instructions, that you can do per year. Now, this 10 to the 50 is roughly 2 to the 10 is ((Refer Time: 20:21)) let us say is certainly less than equal to 2 to the 4 times 50. That is 2 to the 50 for the 200. So, now you can I just divide this by that to check, how many years I am going to take.

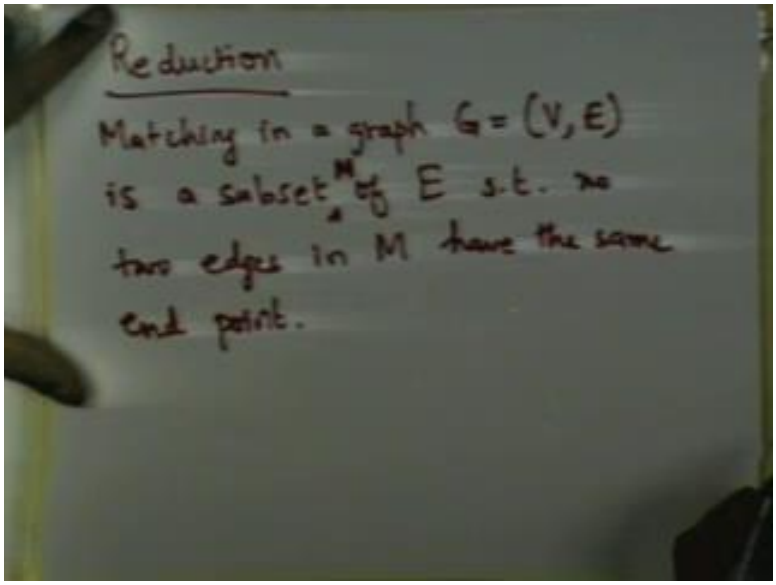
So, the number of years you going to take is 2 to the 1000 minus 200, this is roughly 800 years. You can check, how many particles there are in the universe. This is not very far away from that number of atoms in the universe per year. So, you are going to take a lot of years for this algorithm to finish. This is certainly not what you wanted to do. If the boss figures this out, you certainly fired.

In fact, even if he does not figure this out you are fired. So, what do we do? So, this is a motivation for studying NP completeness. I will give you a recipe, using which you can hopefully save your job. If your boss is going to be a little bit intelligent. So, the first part of the course, which you did design of algorithms, was to help you get a job. The second part that we do theory of NP completeness will help you quickly.

So, our final objective will be to show that, this problem is among hard problems in this world, what do I mean by hard? It means that, so far nobody else has been able to find the solution to this problem. Well, the boss can say I mean this problem, just come up with our company. You know other companies handle other problems, why are you saying that, this is not been solved before.

Well you can in fact, say that if somebody manages to solve this problem. Then, a lot of other problems, which will lot of other people have not been able to solve can also be solved. So, this is the kind of statement we would like to make. So, before we formally look at this notion of NP completeness. We will lead a few other Fonda's, few other notions that we need to imbibe. The first one is the concept of reduction.

(Refer Slide Time: 22:59)

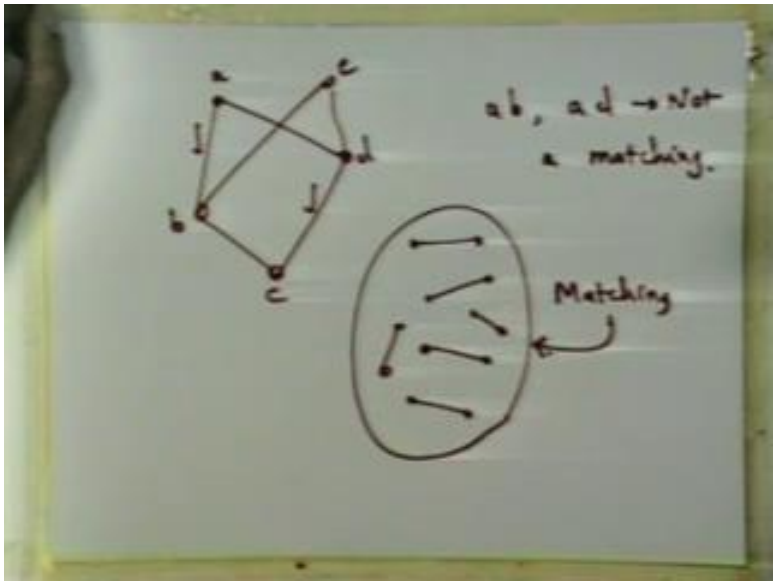


So, we will first look at an example and we will see what reduction really means. So, reduction really means this, supposing you are having a library. So, you have a library where large number of problems have been very well coded by very good programmer. Now, given a new thing that you want to code, it would be very nice if we can use sub routines from the library in your code. Rather than, re invent the wheel and try and do all this by yourself.

This is all that is to reduction. How to efficiently use code for other algorithms to generate new algorithms. So, let us look at an example. This is not very easy example, it is not very hard either. So, before we start a few definitions, which we will recall

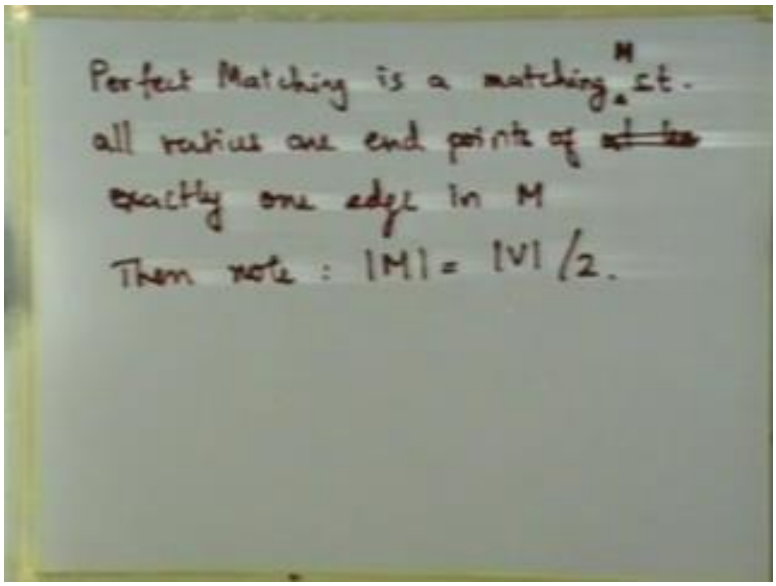
hopefully you have seen this before. The first thing is notion of a matching, in a graph G equals V E, V is the vertex set, E is the edge set. A matching in a graph is a subset of the edge set is a subset of E. So, let me call this M, the subset M of E, such that no two edges share the same vertex. Such that, let me write this down no two edges in M have the same end point, let us look at an example.

(Refer Slide Time: 25:21)



So, let us take a graph, let us say this is a graph. Then, here is a matching, the ticked edges form a matching. If I look at these two edges, they do not share a end point. The end points of these edges are these two, the end point of this edge is these two they do not share any end point. For instance, let me label this a, b, c, d, e. if I take a, b and a, d these do not form a matching, this is not a matching. Essentially one set of edges. So, that they look like this in a graph, they do not share an environment, this is what matching is... The next definition we need is that of a perfect matching.

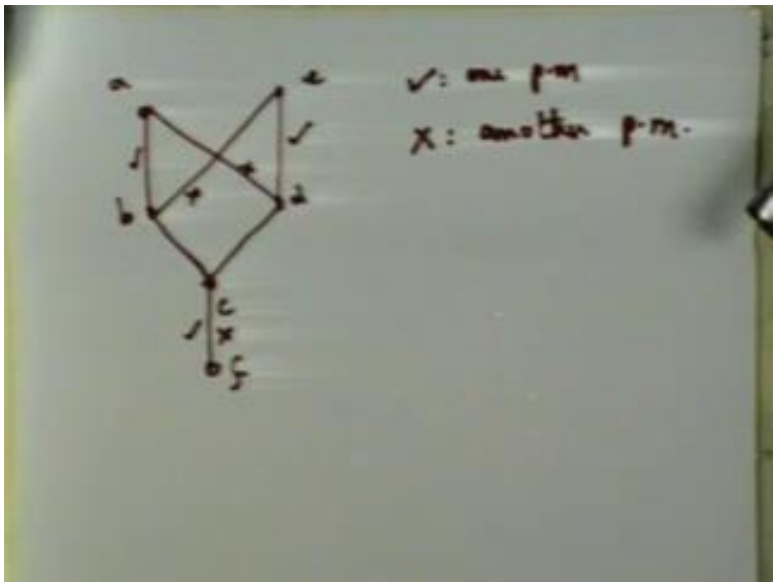
(Refer Slide Time 26:40)



We will say that a matching is perfect, if all vertices in the graph are end points of at least one edge. This is a matching such that, all end points. A perfect matching is a matching, such that all vertices are end points of at least exactly one edge in M. So, it must be matching M such that all vertices are end points of exactly one edge. So, the size of a perfect matching is just half the size of the vertex set. So, then note that size of M is size of V by 2.

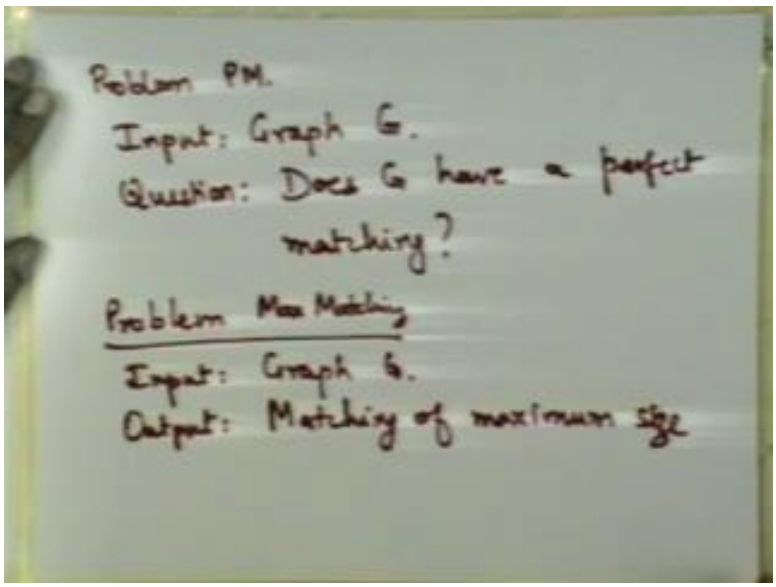
So, this is a perfect matching, let us go back to this old graph of ours, do you think that this graph has a perfect matching ((Refer Time: 28:30)). The answer is no, well there are five vertices. So, if I take a matching with two edges I can cover four of these vertices, I cannot cover five. And I cannot take three edges, because then two of these edges will share an end point. So, this graph does not have a perfect matching, but for instance if I change this graph slightly.

(Refer Slide Time: 29:04)



Let us see I have a, b, d, e, c let me add one more, let us say f. Now, this graph has a perfect matching. In fact, it has many at least two perfect matching's, may be two perfect matching's, here is I take these three edges a, b, e, d and c, f is a perfect matching a d, e b and c f is also a perfect matching. It ticks for one perfect matching and the crosses form another perfect matching, this graph ((Refer Time: 29:55)). So, now that we have defined these two problems. Here is a problem that we would like to solve. So, there are two problems.

(Refer Slide Time 30:20)



So, here is let me say one is problem perfect match PM is for perfect matching. The input is a graph G and the question you ask, does G have a perfect matching. This is your first problem, the input is a graph G does G has a perfect matching. The other problem we are going to look at this, it is very closely related to the first one. So, this is maximum matching, the input is the same, input is a graph G. The output I need is a matching of maximum size.

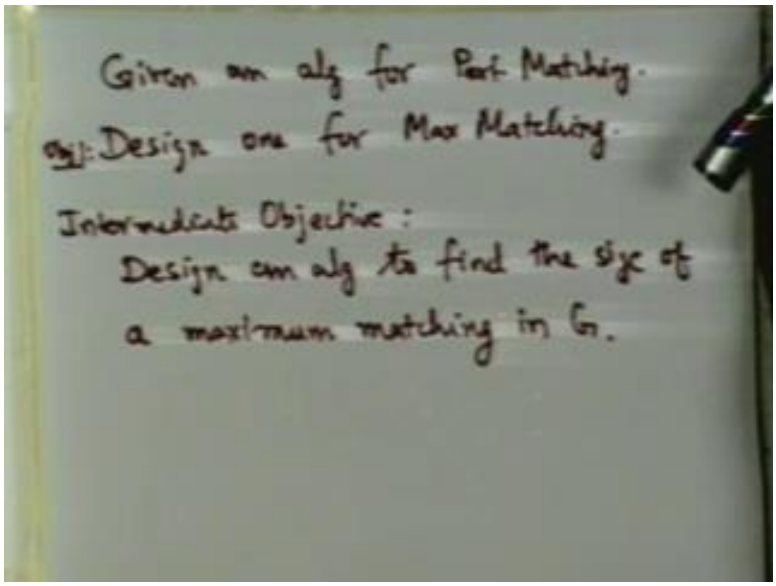
So, let us look at these two problems. For this problem I only need to say yes or no. Ii just want to say whether the graph is perfect matching or not. We actually need an output, which is I need to know the I need edges, which will represent in some matching of maximum size. We have seen that a matching a graph could have more than one matching's of maximum size, any one of them will do.

Now, supposing in your sub routine library, you know you have sub routine coded for this for problem maximum matching which. So, given you have an algorithm, so that when you feed in a graph G, it will output a matching of maximum size. Can we use this to solve all this problem, the answer is yes. Just feed in the same graph G, look at the matching.

If it has V by 2 edges, V is the number of vertices. If this has V by 2 edges, then the graph has a perfect matching. If it is less than V by 2, then the graph does not have a perfect matching. So, given a algorithm for this second problem, very easy to construct one for the first problem. How about the other way around which is given.

Supposing, this sub routine library has an algorithm for this, which is given in graph G, it will tell you whether it has a perfect matching or not. And now, you want to output matching of maximum size. Your job is to construct an algorithm, which does this. And you would like to use this algorithm, effectively. Because, this is a very good implementation and you would like to use this effectively. Let us see how this is done. So, I hope the objective is clear, let me write this down.

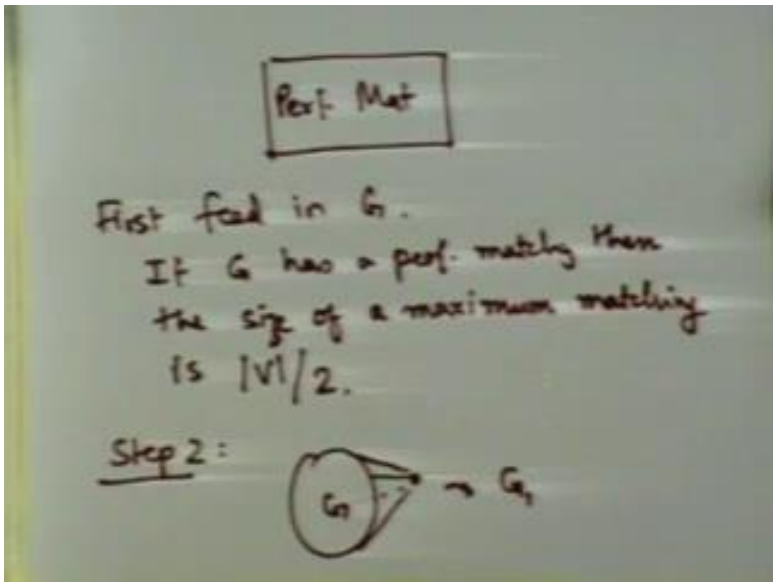
(Refer Slide Time: 34:00)



So, we are given an algorithm for a perfect matching. And we want to design one for max matching. This is what we want to do? Well, how do we really use this algorithm, that is the crucial question? In fact, let us take something which is sort of intermediate between these two, how about time to find the size of the maximum matching. We will look at this problem maximum matching ((Refer Time: 34:54)).

The output should be edges, set of edges in a matching of maximum set of size. That is what the output should be. Now, supposing we only want to know the size of the maximum matching in the graph, can we solve this let me write this problem down and will try and do this. So, this is objective 1 and intermediate objective is this design and algorithm to find the size of a maximum matching in G. So, we are given an algorithm for perfect matching, you want to design an algorithm to find the size of the maximum matching in G, this is how you do it. So, you take this graph G.

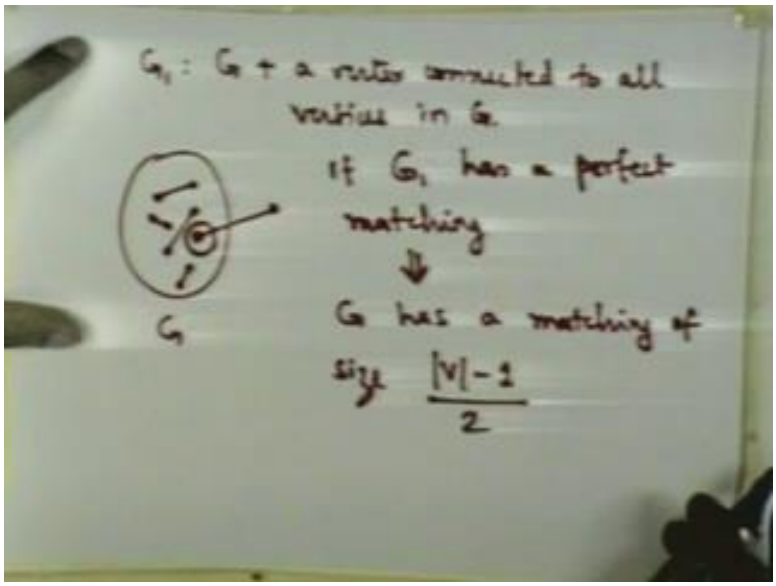
(Refer Slide Time 36:14)



So, here is your algorithm that does perfect matching. Now, you first feed in G. And ask whether it has perfect matching? If G has a perfect matching, then you know the size of the maximum matching is hopefully. It is if G has a perfect matching, then the size is V by 2, where V is the number of vertices in the graph. So, let me write this down, if G has a perfect matching, then the size of maximum matching I is a V by 2.

So, supposing, it says that G does not have a perfect matching now what you do? Well, what you do is this you take here is G. So, step 2, so you take G add a vertex, this is a new vertex connected to all vertices in G. Now, call this graph G 1, now you feed G 1 into this algorithm for perfect matching. You have taken G, you added vertex, which is adjacent to every other vertex in G. And now you feed this into this perfect matching. Again two things can happen, either it can come back and say it has perfect matching or it can say no. Let us see what happens in both cases.

(Refer Slide Time 38:48)

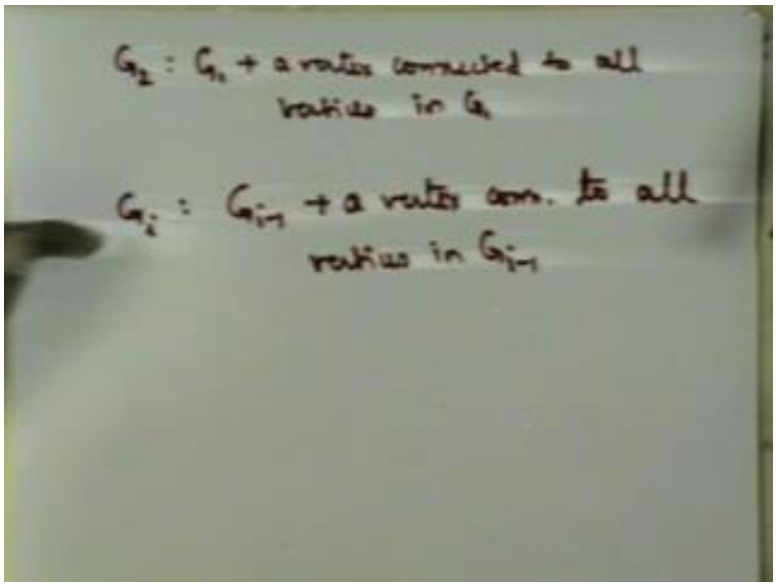


So, G 1 is G plus a vertex connected to all vertices in G, that is what you want. Now, supposing G 1 has a perfect matching. Then, what can you say about G. Well, if you look at a perfect matching in G 1, this vertex this extra vertex is will in the matching, it will be connected to something in G. This is the picture here is G, here is the extra vertex and I just going to draw the edges in the matching.

So, this fellow will be connected to something here in this matching. The other vertices in G are all match somewhere over the other. This is if G 1 has a perfect matching, then this is the figure, this is how it looks like, which means except for this vertex. If I remove this vertex out, the rest of the graph is a perfect matching, which means there is a matching of size, this implies that G has a matching of size V minus 1 by 2.

And this is the size of the maximum matching really. We know that, because V did not have a perfect matching, it does not have a matching of size V by 2. On the other hand, it has a matching of size V minus 1 by 2. This is if G 1 has a perfect matching, what if G 1 does not have a perfect matching. If G 1 does not have a perfect matching, then we know that G does not have a matching of this type. So, then we create graph G 2.

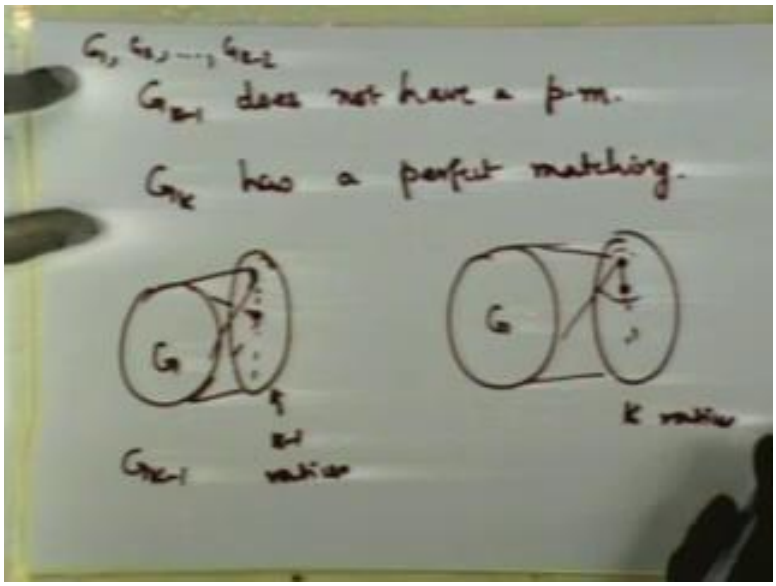
(Refer Slide Time: 41:05)



So, how do we create G 2 well, similar you take G 1 plus a vertex connected to all vertices in G 1. Now, you ask G 2 has appropriate matching and so on. Then, you form G 3 and ask whether G 3 has a perfect match. And you stop as soon as the perfect matching fellows says yes. So, let me write the generic step. So, you get G i the graph G i minus 1 plus a vertex connected to all vertices in G i minus 1.

Well, this is how you get G i. So, now you look at G 1, G 2, G 3 and so on. And ask whether G 1 has a perfect matching, G 2 has a perfect matching and so on. So, let us say k is the first index, where the perfect matching algorithm says yes. So, which means...

(Refer Slide Time: 42:32)



So, assume that G k minus 1 does not have a perfect matching and G k has a perfect matching. Now, what is G k minus 1 and G k look like. So, here is my original graph G for G 1 I added one vertex with this connected every single region this is G 1, G 2 I added two vertices. Now, this is connected to this and also connected to everything and so on. So, G k minus 1 I have k minus vertices here, they are connected through each other and they are also connected to everything in G.

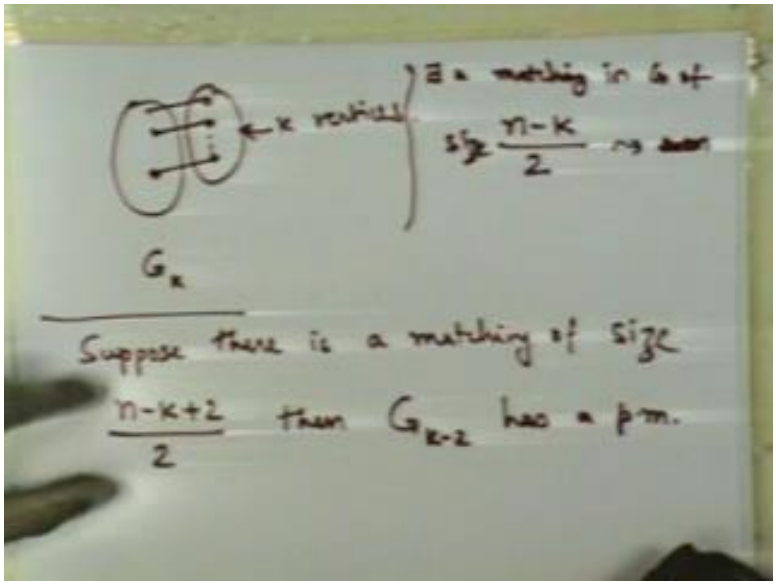
Similarly, this is G k minus 1 and G k looks like this I have the graph G and k vertices here. These are connected to each other and they are also connected to each other k vertices which are connected to everything each other. Now, this does not have a perfect matching and this has a perfect matching, what does this mean. Now, let us look at the perfect matching here.

Now, actually I must say that G 1, G 2 all the way up to G k minus 1 does not have a perfect matching. So, k was the first time that we encountered a graph with perfect matching G 1, G 2, G 3 up to G k minus 1 and does not have one and G k has one. So, let us look at G k. So, what can a perfect matching look like, can it have a edge like this, the answer is no.

Why is that, supposing it has an edge like this in the matching, you remove these two and this graph now looks like G k minus 2 and this as a perfect matching. But, we know that

G k minus 2 does not have. So, all edges in the matching must be from vertices here to vertices inside G. So, here is the picture here is G.

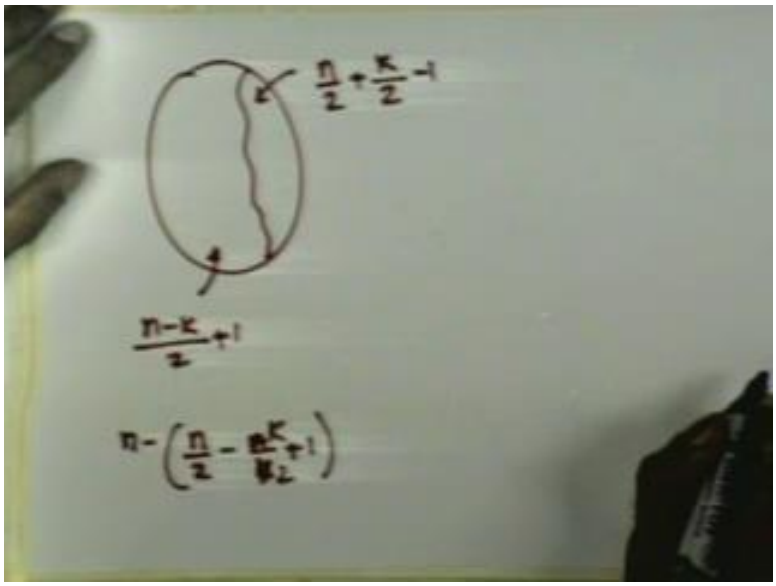
(Refer Slide Time 45:07)



So, this is G k these are k vertices, these k vertices are matched with some k vertices in G. There are n minus k other vertices in G, which are matched amongst themselves. So, there is a matching inside G of size n minus k by 2. So, there is matching in G of size n minus k by 2. And we would like to observe that, there is no smaller, there is no matching of size larger than n minus k by 2. This is the largest matching that you can have, now why is that so.

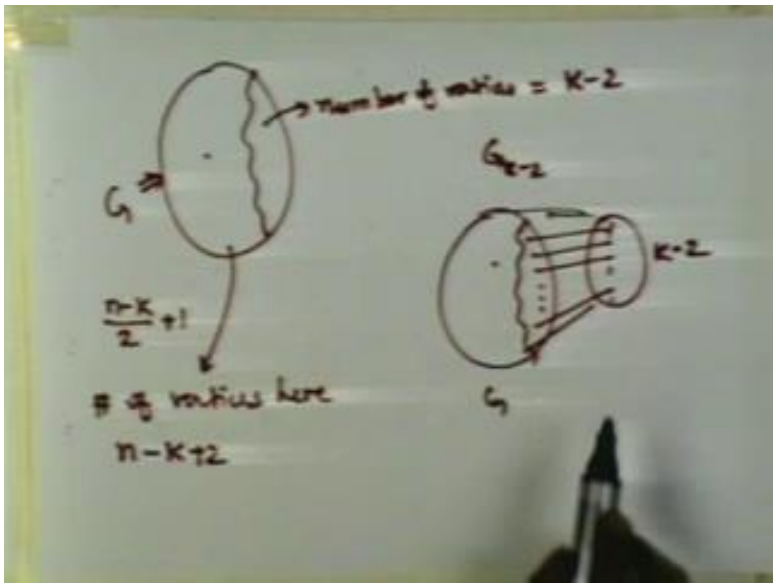
Supposing, first this has to be even m minus k has to be even, because this is an integer right. So, if there is a matching of size larger than this. That means, so suppose there is matching of size, let us say n minus k plus 2 by 2 which is nothing but, n minus k by 2 plus 1 one more edge. So, supposing there is a matching of this size. So, the claim is that one of the previous graph. In fact, then G of k minus 2 has a perfect matching. So, this will be a contradiction. Supposing there is matching of size larger than this, then G k minus 2 will have a perfect matching. So, we will see this.

(Refer Slide Time 47:26)



So, here is G and here is a subset which is of size n minus k by 2 plus 1. So, here I have n minus these many which is... So, n minus n by 2 plus k by 2, that is this size, which is n by 2 minus and minus plus k by 2 minus 1.

(Refer Slide Time 48:20)



So, let us look at this let me just refresh ((Refer Time: 48:24)) we are here supposing there is a matching of size n minus k plus 2 by 2. And we have to show that G k minus 2 has a perfect matching. So, this is G, so it has a matching of size n minus k by 2 plus 1.

So, the number of vertices in the matching is twice this, so the size, so number of vertices here is twice this, which is n minus k plus 2.

So, the number of vertices here is n minus, so which is k minus 2. This portion has a perfect matching, which was our assumption. So, let us look at G k minus 2. So, G k minus 2 had G, this is G this in fact, is G. And I had k minus 2 other vertices, which were connected to everything in G, and also amongst themselves, if this was G k minus 1.

What I do is this, I take this matching in this portion. As if and each of these k minus 2 vertices I match to the other size. You can see that, this gives perfect matching. Matching on this portion remains the same as in this portion here. And these k minus 2 vertices I match to the other side. So, this shows that G k minus 2 in fact, has a perfect matching ((Refer Time: 50:20)).

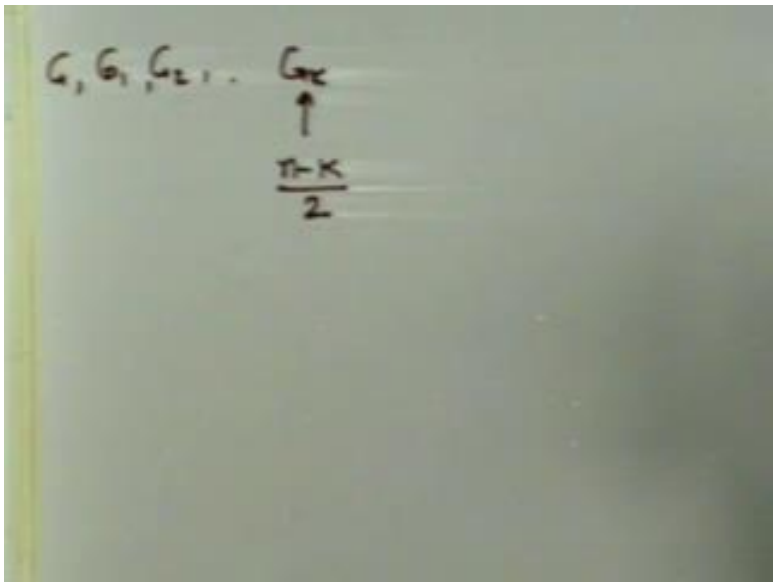
So, let us recap we were given a algorithm for perfect matching. And the objective the intermediate objective was to design a algorithm to find the size of the maximum matching in G. And what we do is this? ((Refer Time: 50:38)). We define graphs G i, where G i's G i minus 1 plus the vertex connected to all vertices in G i minus 1 ((Refer Time: 50:49)).

We find the first graph we look at G 1, G 2 and so on. And we find the first graph G k which has a perfect matching, which means G 1, G 2, G 3 and so on up to G k minus 1, they do not have a perfect matching. G k has a perfect matching ((Refer Time: 51:05)). Then, we claim that there is a matching in G of size n minus k by 2 and no larger.

So, the size is the largest matching is n minus k by 2. This shows that given an algorithm for finding the perfect matching. I can design an algorithm for finding, the matching of maximum size in a graph by calling the other algorithm repeatedly I do not call it too many times. I call it at most k times here. You can actually do it much faster, you can call it faster than k, so in the following sense.

This k can it can go down all the way up to 4, 5 up to a constant, which means you can call it about n times. So, we have shown that given an algorithm to find the perfect matching in a graph. I can design an algorithm that finds the size of the maximum matching.

(Refer Slide Time: 52:19)

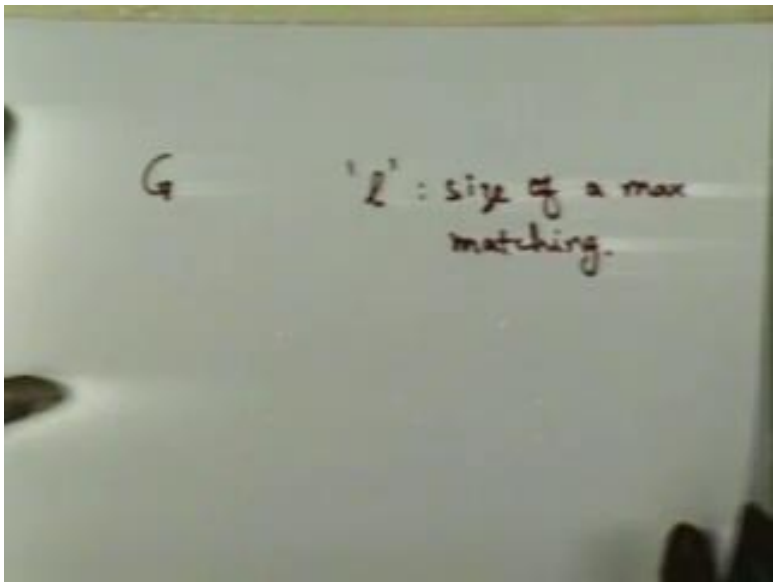


The way you do it is you look at graphs G, G 1, G 2, and so on. Find the first graph G k which has a perfect matching. And the size of the maximum matching, then is n minus k by 2. If k is 0, it is G and the size of the maximum I think n by 2 and so on. If G k is the first time you get a perfect matching, the size is n minus k by 2. In the worst case, you would call the perfect matching algorithm n times. I would call G 1, G 2, G 3 and I can go all the way up to n.

Now, there is a smarter way to do this using binary search, which I will let you do, which let you figure out this k, using only log n calls to the algorithm for perfect matching. It resembles binary search and is an exercise for you. ((Refer Time: 53:17)) essentially it shows that by calling this algorithm at most n times I am able to determine at least the size of the maximum matching in the graph.

But, let us go back now, the objective we had was to find this perfect matching. So, here now I have full filled this intermediate objective ((Refer Time: 53:39)). I can design an algorithm to find the size of the maximum matching, I want to actually find the edges in the maximum matching how do I do this. Now, here is the trick, you take this graph G.

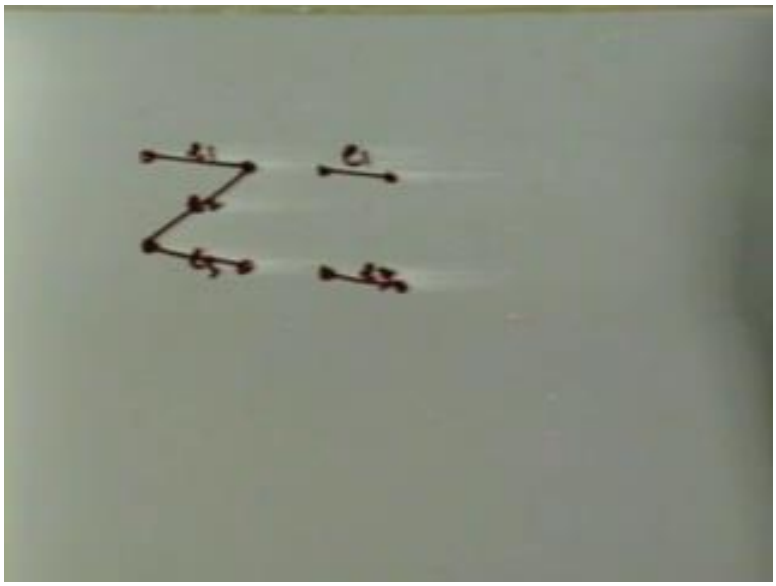
(Refer Slide Time: 54:01)



So, I take G I find the size of the perfect, size of the maximum matching. So, let us say the size is l, the size of a maximum matching. Now, what I do is, pick any edge in G and remove it, remove a edge from G. And now ask the same question, what the size of the maximum matching? If it remains l, then I throw away this edge and I concentrate on the graph at remains. If it is not, if it decreases then I put this edge back. This edge is part of the maximum matching.

So, I leave this edge I look at the edges e 1, e 2 let us say up to e m 1 by 1. When, I look at a edge I remove it and I ask, if the resultant graph has a matching of size l. I find the size of the maximum matching the resultant has. If it is l, then I throw away this edge. Otherwise, I put this edge back by the time I am through with all these edges. But, what I will be left with is a graph which will be just a matching. It will just be a matching of size l. Let us do an example. So, let us take a simple graph.

(Refer Slide Time: 55:43)



So, here so let us take a simple graph e 1, e 2, e 3. The size of maximum matching here is 2 I remove e 1 and I ask what is the size? Now, the size is fallen by one the size is one. So, I put e 1 back I remove e 2, I ask what is the size? Size is 2. So, I throw away it, at this point the graph looks like this, because I have thrown away e 2. Then, I ask for e three I remove e 3 and the size falls by one. So, I put e 3 back. So, I am left with e 1 and e 3 and when I look at the graph that remains is the matching.

In fact, it will be a matching of size l, apart from the matching of size l. If there is any other edge, you know when you remove this edge, the graph will still have a matching of size l. So, you will throw this edge away. So, at the end of this procedure, what you will be left with is the matching of size l, which is what you are looking for.

So, this completes our objective which we started out with. We were given a algorithm for perfect matching. And we wanted to design an algorithm for maximum matching. What we did first was we designed algorithm ((Refer Time: 57:04)) to find the size of maximum matching. And using this we were able to design one for maximum matching. So, we had this in our sub routine library, we added this to the library by using this algorithm. And once we had this in the library it was easy to design a algorithm here. We started with this and we ended.