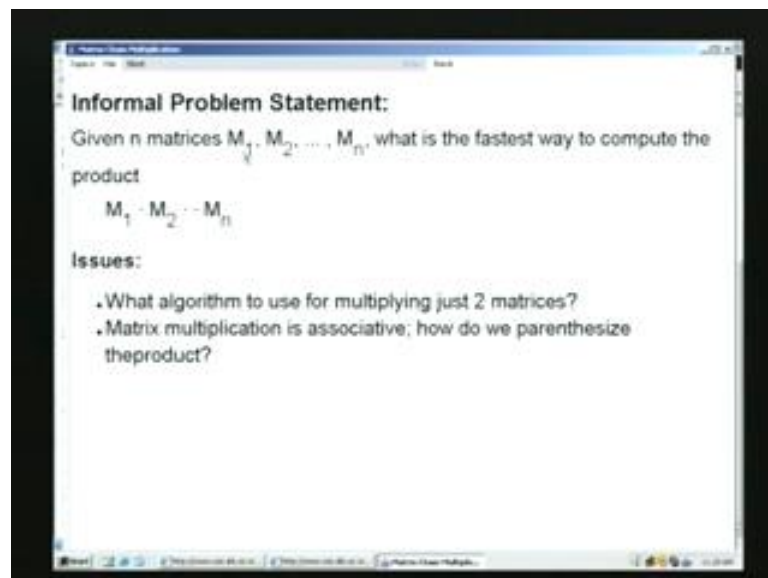


Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 20
Matric Chain Multiplication

Welcome to another lecture on Design and Analysis of Algorithms. Our topic for today is Matric Chain Multiplication. Let me begin with an informal statement of the problem that we have.

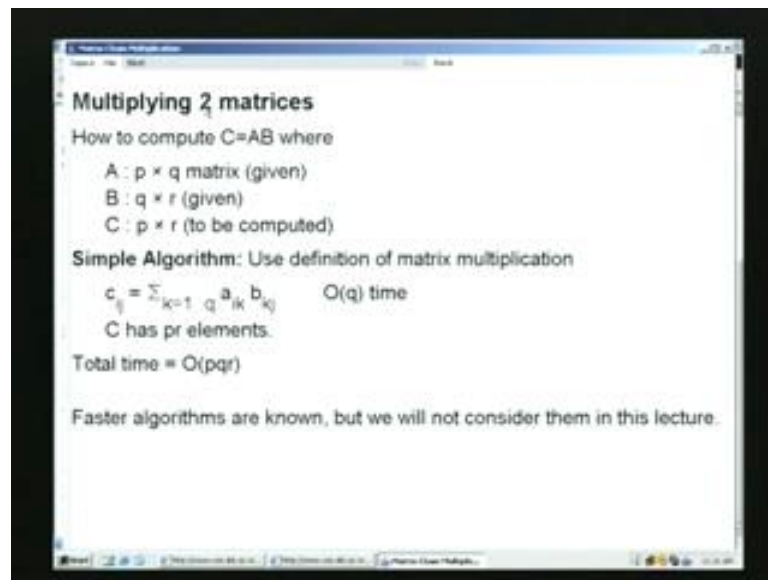
(Refer Slide Time: 01:02)



So, our problem is something like this. We are given n matrices. Let us call them M_1, M_2, \dots, M_n . And we would like to multiply them and form this product. So, this is the usual matrix product. We would like to do this as fast as possible. Here are some of the issues involved in doing this. If the first issue is what algorithm should be use, when we just multiply to matrices together, because we are going to use this algorithm repeatedly. That is how we are going to solve this problem today.

The second issue is since matrix multiplication is associative. How do we parenthesize this product. Because, that will, it will be clear zone. That will also affect the time required to do the entire to perform the entire product. So, let us look at the first issue.

(Refer Slide Time: 01:55)



So, let us say we want to compute C equal to A times B. Where, A is a p by q matrix. B is q by r matrix. A and B are given to us. And C is a p by r matrix, which is to be completed. The simplest algorithm for doing this is to use a definition of matrix multiplication. Matrix multiplication is defined as C_{ij} is equal to summation over k, k going from 1 to q of a_{ik} times b_{kj} . Computing the single term such as C_{ij} will require O of q time. C has p r elements overall C is a p by r matrix. So, it has p r elements. So, the total time will be O of p q r.

Now, terms out that there are faster algorithm possible. But, we are going to strict to this algorithm for simplicity. So, we will assume for the rest of the lecture. That if I want to just multiply 2 matrices together my time is going to be O of p q r or to wake it even simpler. I am going to just say p q r is the time required for multiplication of 2 matrices. p by q matrix by and q by r matrix.

(Refer Slide Time: 03:13)

Associativity

Example: $n=3$: Need to compute $M_1 \cdot M_2 \cdot M_3$

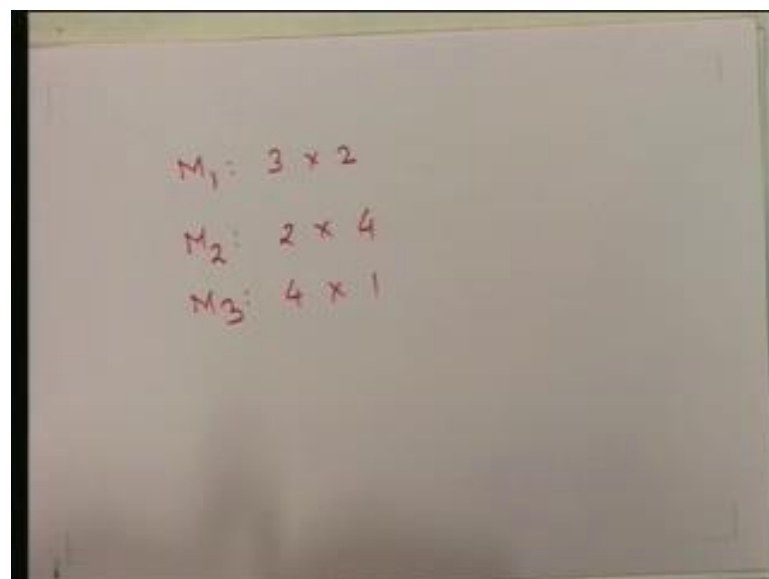
Which is better: $M_1 \cdot (M_2 \cdot M_3)$ or $(M_1 \cdot M_2) \cdot M_3$?

Time[$M_4 = M_2 \cdot M_3$] = $2 \times 4 \times 1 = 8$
Time[Ans = $M_1 \cdot M_4$] = $3 \times 2 \times 1 = 6$
Time[$M_1 \cdot (M_2 \cdot M_3)$] = $8 + 6 = 14$

Time[$M_5 = M_1 \cdot M_2$] = $3 \times 2 \times 4 = 24$
Time[Ans = $M_5 \cdot M_3$] = $3 \times 4 \times 1 = 12$
Time[$(M_1 \cdot M_2) \cdot M_3$] = $24 + 12 = 36$

The second issue concerns associativity. Let me explain this with specific example involving only 3 matrices. So, say n is equal to 3 over here. Now, we need to compute the product M_1 times, M_2 times, M_3 . There are two ways to do this. One possibility is multiply M_2 and M_3 first. And then multiply that with M_1 . Or multiply M_1 and M_2 first. And then multiply that with M_3 . Let us, take a specific example.

(Refer Slide Time: 03:55)



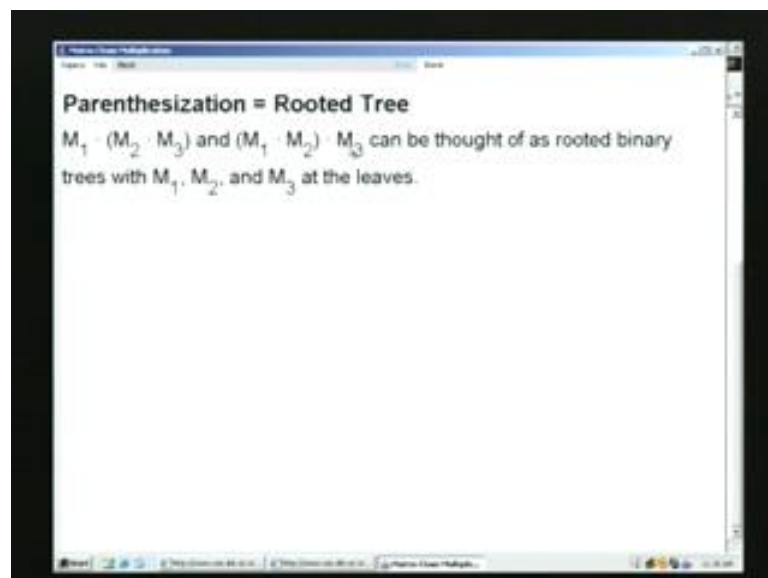
So, let us assume that M_1 . M_1 is a 3 by 2 matrix. M_2 is a 2 by 4 matrix. And M_3 is a 4 by 1 matrix. So, now if you want to multiplied according to this manner ((Refer Time:

04:23)). Then here is what will happen. So, we will first compute matrix M_4 , which is M_2 times M_3 . So, M_2 was 2 by 4 matrix. And M_3 was a 4 by 1 matrix. So, the product will be following over formula $p \times q \times r$, it will be 2 times 4 times 1, or it will be 8. Now, given M_4 our final answer can be computed just by multiplying M_1 with in.

M_1 is a 3 by 2 matrix. And M_4 we calculated ((Refer Time: 04:58)) 2 by 1 matrix. So, this will require time 3 by 2 by 1 or 6. So, this will require 6 matrix multiplication, 6 scalar multiplication this will require 8s scalar multiplications. Then the total number of multiplication required will be 14. Another possibility, which I have indicated over here, is to first form M_1 times M_2 . So, we compute M_5 equals M_1 time M_2 .

Remember, that M_1 was in 3 by 2 matrix. And M_2 was a 2 by 4 matrix. So, the total time over here, to calculate M_5 is equal to 24. The answer now, can be obtained by multiplying this M_5 which is just this, which we calculated by M_3 . Now, M_5 in the product was 3 by 4. So, it is P and Q are 3 and 4. M_3 is 4 by 1. So, Q by R is 4 and 1. So, this product requires time 3 times 4 times 1 or 12. The total time for this way of doing things is therefore, 24 plus 12 which is equal to 36. So, clearly this first way is better.

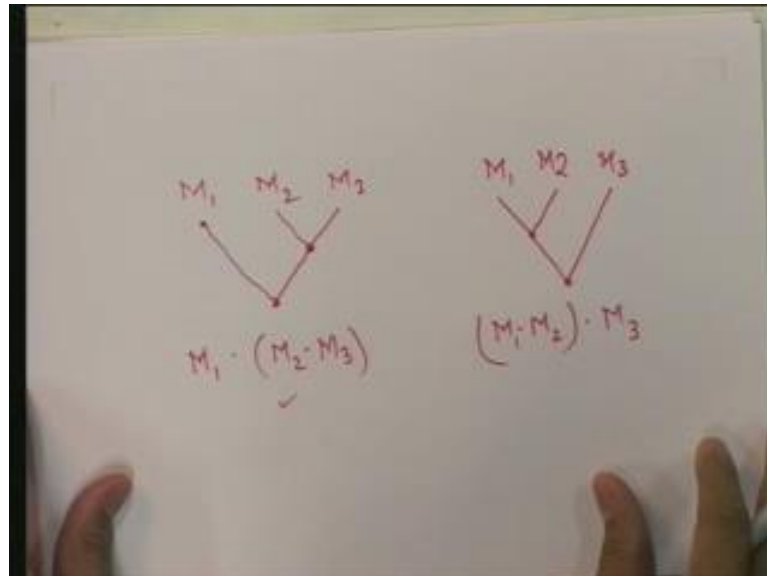
(Refer Slide Time: 06:18)



The problem now is that, if we are that we can take our product. And we can parenthesize it in this way or we can parenthesize it in this way. We could speak of

parenthesization. But, I would like to point out that these parenthesization can be thought of as rooted trees. So, let me explain that.

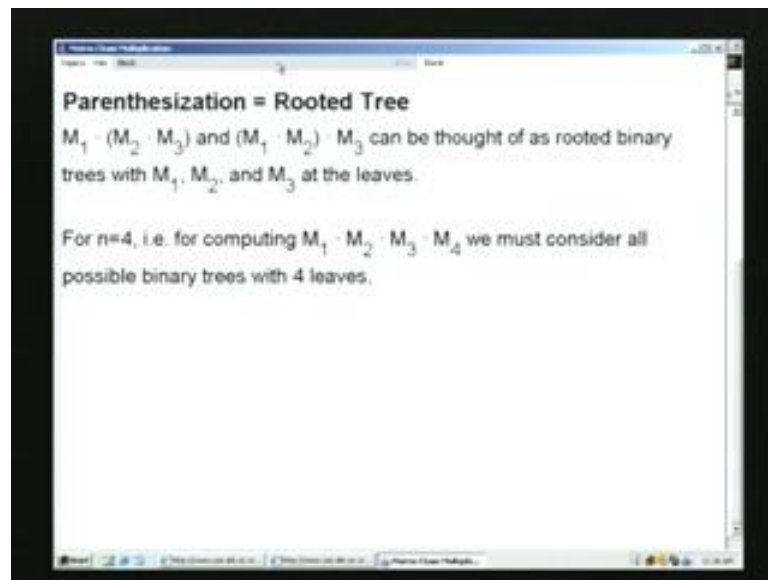
(Refer Slide Time: 06:43)



So, here are our matrices to be multiplied M_1 , M_2 , M_3 . So, if I want to first form the product M_2 times M_3 . Let me represent it in our usual fashion by drawing a vertex over here. And this vertex represents their product. The next thing I do is, I take the product of these two. And this is my final answer. So, this corresponds to the parenthesization M_1 times M_2 times M_3 . Alternatively, I could also have done something like this. So, I could take product of M_1 and M_2 first. So, this is the first way of doing things.

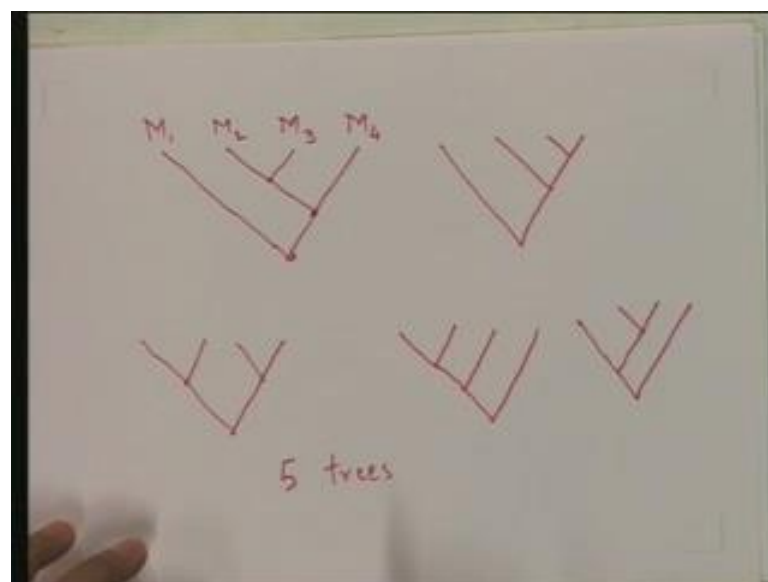
This is this is what I do first and then I take the product of those. So, the parenthesization corresponding to this is M_1 times M_2 whole times M_3 . So, this is this we have doing things which we had early indicated earlier. In this particular example, we found that this ((Refer Time: 07:49)) superior. But, of course, with other choices of the matrix to mention, other way could be superior this could be superior. We want multiply M matrices ((Refer Time: 08:04)) just 3. So, let us do one more example, what if we have 4 matrices.

(Refer Slide Time: 08:07)



So, in this case, we have addresses M_1 , M_2 , M_3 , M_4 . So, the parenthesizations here, will correspond to 4 trees with 4 leaves. So, one possible parenthesization, which I might indicate is something like this.

(Refer Slide Time: 08:32)



So, say for example, I could compute say the product of M_2 and M_3 first. Then, maybe I compute product of M_3 and M_4 next. And then finally, I compute this. So, this is my, this is the tree. Of course, there are many such trees possible. And in fact, in this case let me just write down all the values all those. So, say for example, here is another tree. So,

here is another tree, in which we first multiply and 3 and 4. Then multiply that with M 2 and then with M 1.

But, that is not only one here is another. And then, we also have things like and also. So, in this case, it turns out that there are 5 trees in which M 1, M 2, M 3, M 4 can be the leaves. And therefore, 5 ways of parenthesize in this product. As you saw to go from 3 to 4 we went from 2 to 5. If you keep doing this, you will see that the number of tree is grow is very fast. In fact, you can prove you should be able to prove that it grows exponential. So, in the pointers that we want to pick a tree from one of these possible trees, and we want to pick the 1, which minimize this the total time taken. So, we are now ready to give a formal statement of the problem.

(Refer Slide Time: 10:31)

Formal Statement of Problem:

Input: $d[0..n] : d[i-1]$. $d[i]$ respectively give the number of rows and columns of matrix M_i

Output: Rooted binary tree with M_1, \dots, M_n as leaves representing the least time parenthesization.

Time required by a tree =

Time for left subtree + Time for right subtree

+ Time to multiply of the result of left subtree and result of right subtree.

Note: If left subtree has first i leaves, then result of left subtree is a $d[0] \times d[i]$ matrix, and result of right is $d[i] \times d[n]$ matrix.

Time = Time(Left subtree) + Time(Right Subtree) + $d[0] d[i] d[n]$

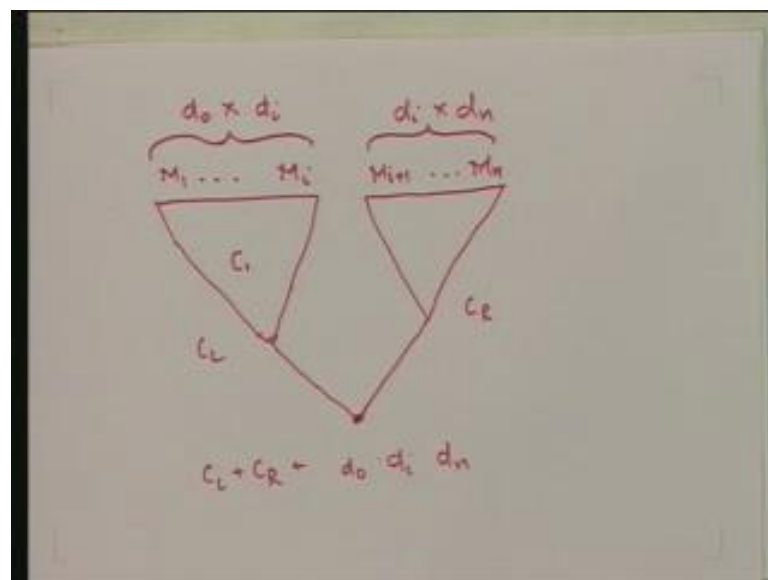
So, our input consist of an array d zero through d n. So, this is going to represent the dimensions of the matrix. So, d i minus 1 and d i respectively will give the number of rows and columns of matrix M i. So, this means that the columns of the number of rows of matrix M i plus 1, will be d i. So, which will be equal to the number of columns of matrix M i. Which is as I should be, because only then can be think of the product M i times M i plus 1.

So, although we have M matrices, which are being multiplied and this is supposed to represent the dimensions of N matrices. We only have a, have N plus 1 numbers over here. Our output is suppose to be a rooted binary tree with M 1 to M n as leaves. And

this whole thing should give us the least time for computing the product. Let me now, state as state again. That if you give me a tree, I can estimate the time required for the tree. And the algorithm here is actually fairly straight forward.

So, if we give me a tree, then the time required for the left sub tree can be estimated recursively. The time required for the right sub tree can be estimated recursively. And then, I just add up the two times. But, that does not finish the time the total time. I have to take the product represented by the left tree. The product represented by the right tree and I have to multiply them together. So, I need an additional term over here. So, just to clarify suppose, we have we consider a tree, in which the left sub tree has I leaves.

(Refer Slide Time: 12:28)



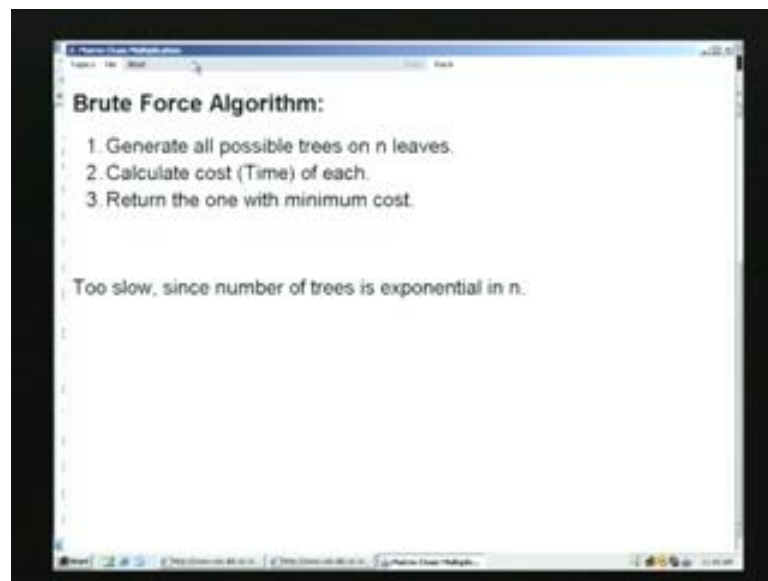
So, say M_1 through M_i belong to the left sub tree. And M_{i+1} through M_n are leaves for the right sub tree. And so we take the product over here. So, this has some cost let me call it C_L and let me call the cost of this C_R . So, what is the cost of this entire tree? Well, So, the cost is going to be C_L plus C_R plus computing the product of these matrices with the product of these matrices. But, what is the product of these matrices, what is the dimension of the product of these matrices.

So, that is simply the number of rows in this matrix, which is d_0 , times the number of columns in this matrix. So, let me write over here. So, this are here, this product is going to be d_0 times, the number of columns over here. So, times d_i , this product is going to be d_i times d_n . So, as a result, what we are going to get is the total time or the time to

computer this lost product, which is the only term remaining over here is d_0 times d_i times d_n .

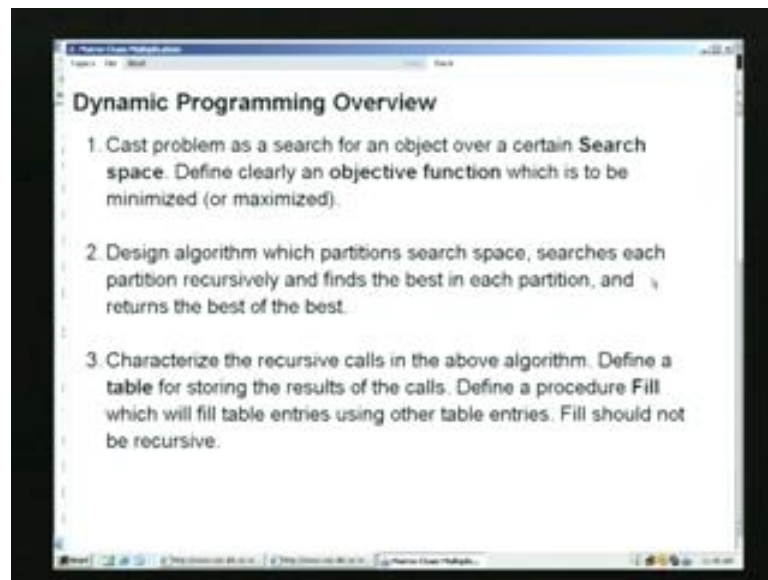
So, stated here, the time taken is, the time for the left sub tree, time for the right sub tree plus $d_0 d_i d_n$. So, here I was using subscripts. Here, I will just use array accesses, but you know there will be in the same thing. I mean the same thing by them.

(Refer Slide Time: 14:40)



So, how do we solve this problem. So, here is a simple brute force algorithm. So, we generate all possible n leaf trees. And will use, will calculate the cost for each or cost is really the same as time in optimization ((Refer Time: 15:00)) more customer is to think of cost. So, I might use or I will use cost to represent time in the rest of the lecture. So, we calculate the cost of each of these trees. And then, will be return the 1 with the minimum cost. As remarked earlier, this is going to be too slow, because a number of trees is going to be exponential in n .

(Refer Slide Time: 15:22)



We would like to solve this problem or we would like to see, if dynamic programming can be used to design an efficient algorithm for solving this problem. So, let me give a quick overview of what dynamic programming the whole dynamic programming ideas. So, in dynamic programming the first step is to cost the problem as a search for an object over a certain search space. So, this certain search space, we have to define a ((Refer Time: 15:50)) clearly, just to clarify our thoughts.

Then, we also need to define clearly an objective function, which is to be minimized or maximized. In our problem, we have really done this step. Because, the search space is simply the search space is simply the space of all possible trees. And the cost function is the time and that is to be minimized, the objective function or the cost function. The next step is to design an algorithm, which partitions search space. And each partition is searched to find the best, the best element in it or in our case the best tree in.

Now, here is the key idea, each partition is to be searched, but we can do this recursively. So, we find the best element in each partition. And since, we want the best element in the entire space, which is staying the best of these best. Dynamic programming does not stop at this step 2. The last and most important step perhaps or the most unusual steps perhaps is that we are going to characterize, the recursive calls which over made over here.

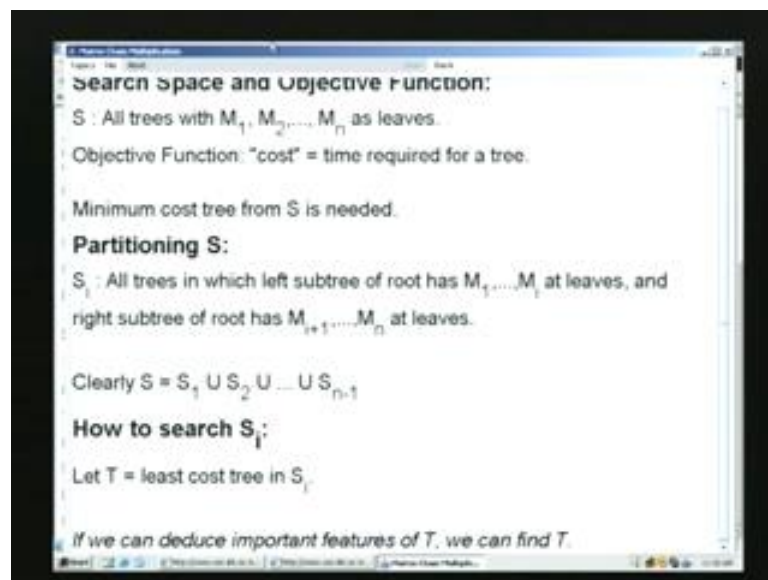
So, in this we will making several recursive calls, we will ask the question. Is there anything interesting about those calls, is there any property that we can identify for those

calls. Or somehow, can be say that the calls are only made on these arguments. So, once we get a characterization, we are going to build the table, in which the results of all these calls are going to be stored. So, we have identified, which are the calls which are going to be made over the lifetime of this algorithm.

And will have an entry in this table corresponding to each such call. Then will define a procedure, which I will call fill, which will be able to fill the table entries using other table entries. The key requirement is that fill should not be recursive function. If we can do this, we will have a dynamic programming algorithm. And if everything works right this will be a fast algorithm. So, let us see how we can make this work. So, the first thing is to look at the search space. So, let us quickly take a look at that.

So, the search space in our case is all possible trees with M_1, M_2, \dots, M_n the matrices at the leaves. Note, that we have not actually been given the matrices. And we do not need them, we do not we are not actually going to calculate the matrix product. We are just going to design scheme for calculating the products.

(Refer Slide Time: 18:50)



So, even though I say, that M_1, M_2, M_n are the leaves. I really mean, so I really mean there are some place holders M_1, M_2, M_n where the matrices can eventually be placed. So, there are these place holders M_1, M_2, M_n which are going to be the leaves. And we are going to consider all possible trees and that going to be our space S . Our object

function is going to be a cost. We call it a cost, because that is something that we want to minimize.

So, our cost is going to be the time required for a tree. And we have already seen, how we calculate this time. So, our goal is to find the minimum cost tree from this S . The next step is to partition this S . Now, there are several ways in which you could partition. If you have so this is the place, where we require some creativity. However, I am going to point out a very natural way of doing this partitioning. So, S is the space of all trees. I am going to define S_i is that sub space.

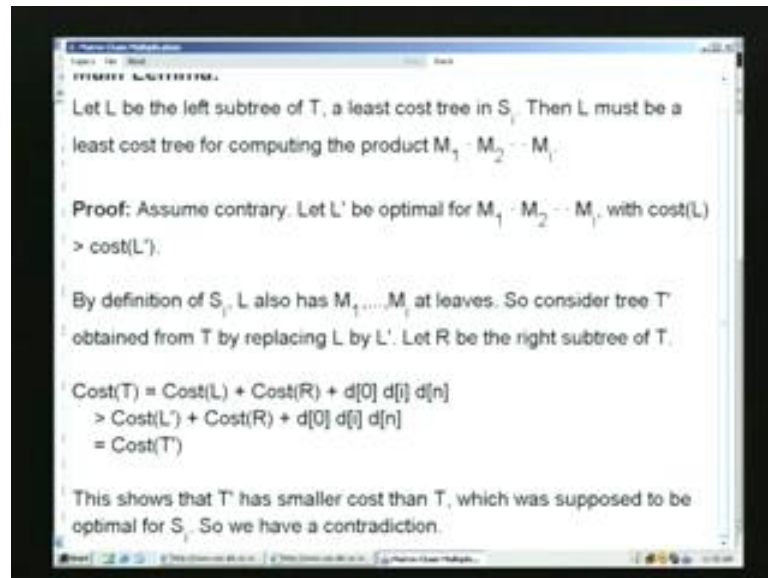
So, that sub space in which consists of all trees, in which left sub tree of root has M_1 through M_i at the leaves. And right sub tree of root has $M_i + 1$ to M_n at the leaves. So in fact, it is going to be a tree which looks like this. So, ((Refer Time: 20:22)) in this picture, I have M_1 through M_i at the leaves of the left sub tree. So in fact, let me call this sub tree L and let me call this R . Now, I have told you, what structure L is going to have, it could have any structure what so ever.

I have told you what structure R is going to have, that could also have any structure. So, all such trees, which have this form R going to be in this sub space S_i . So, I am going to consider as a sub space for each value of i , i going from 1 to $n - 1$. Because, I want at least 1 leaf in this side and 1 leaf in this side. So, if I take the union of these subspaces S_1, S_2 all the way till S_n, S_{n-1} . I will get my original space. So, every tree in this original space is place, somewhere in one of these subspaces.

The next question is how do I search S_i . So, if we can design an algorithm for searching S_i . Then will be able to find the best in S_i . And then, will take the best of the best as we has said earlier. So, whenever we look for something it is usually a good idea to characterize to design some property to define some property of it. So, as I am looking for a tiger in a forest. Well, it will be useful, if I say that a tiger has strips or it is yellow or it is a large animal.

So, if we can define some properties for the best tree that we are looking at. So, let us say T is the least cost tree in this S_i . And if we can define some properties for it, then it will help us in actually finding T .

(Refer Slide Time: 22:18)

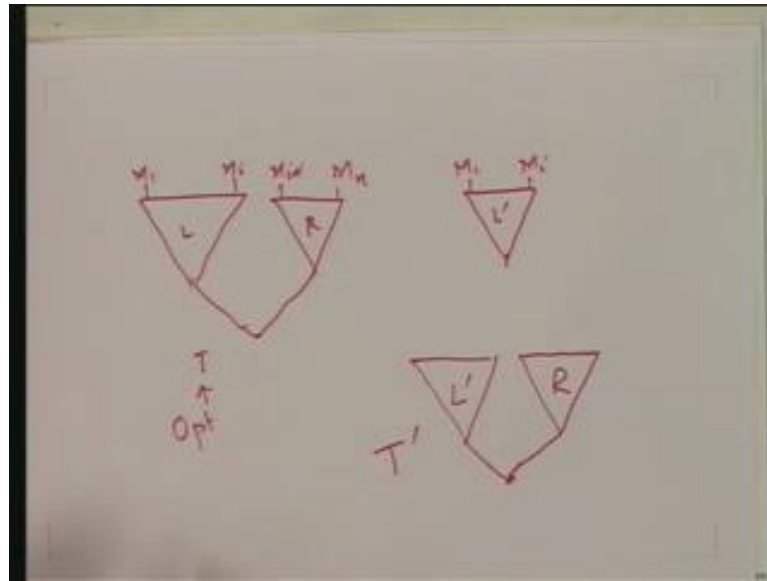


Now, here is the main property of T that we are going to use. And it is actually a fairly natural property. So, the main lemma that we are going to prove is this. So, left L be the left sub tree of T, which is a least cost tree in the side. I am saying a, and not b least ((Refer Time: 22:41)) cost tree, because there could be several least cost trees. So, T is just a least cost tree in our subspace S i. And L is it is that sub tree. The interesting observation or the interesting ((Refer Time: 22:57)) perhaps right now.

Is that, if t is a least cost tree. Then somehow, we expect that L also should be a least cost tree. But, L is only over the L only is computing the product M 1 through M i. And so, our claim is that L must be a least cost tree for computing the product M 1 through M i. And the proof of this is very follows in the standard dynamic programming ((Refer Time: 22:28)) argument. So, we are going to assume ((Refer Time: 22:30)).

So, we will assume that L prime be the optimal or the best tree for this product M 1 through M i. And it is cost sorry and the cost of L is actually, greater than the cost of L prime, which has to be the case, if you want to claim that L is not optimal. So, L has cost strictly bigger the L prime. So, let me draw picture over here another picture.

(Refer Slide Time: 24:02)



So, this is L, this is R and say this is my tree T. Now, what we are just claimed is that there is T L prime. And this has leaves M 1 through M i. And the cost of this is smaller in the cost of this. Of course, L must also have leaves M 1 through M i. And let me write down the leaves over here as well this should be M i plus 1 through M n. Here, is what we are going to do. Here is sort of the standard dynamic programming idea. I am going to construct another tree T.

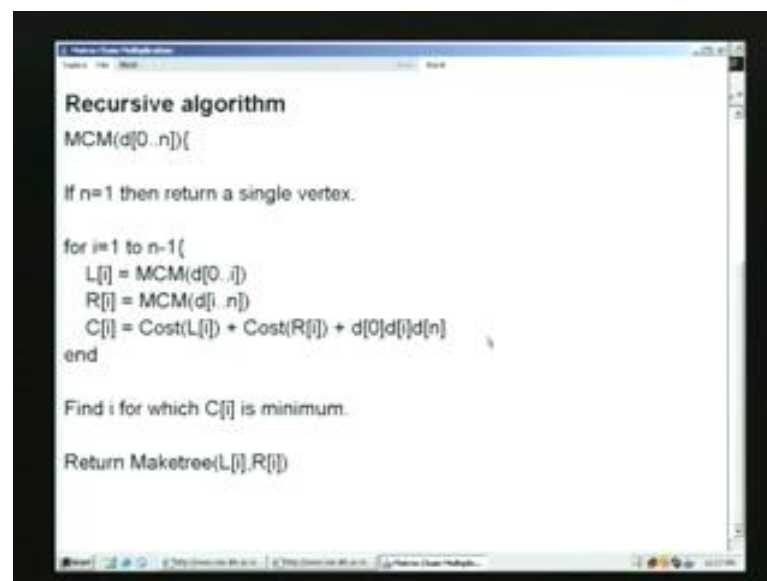
But, this time I am going to place L prime instead of L. So, my tree t is tree T prime is going to looks something like this. So, I am going to have L prime over here. And I am going to retain R as it was. And I am going to connect these two together. So, this is my tree T prime. So, I start, so T is known to be optimal. So, this T is optimal and I do not know anything about this. But, I would like to estimate the cost of this. So, let us look at the cost of T first.

So, cost of T is equal to cost of L plus ((Refer Time: 25:35)) cost of R plus d zero d i d n. So, this just comes from this observation, which we had made earlier. Now, what we know about cost of L, cost of L is bigger than cost of L prime. So, if I substitute L cost of L prime over here. I get that cost of T is bigger than cost of L prime plus cost of R plus all of this. Now, this entire expression really, if you look at thus picture over here, you can see is nothing but cost of T prime.

So, this entire expression ((Refer Time: 26:16)) is equal to cost of T prime. So, what have you proved, we have proved the cost of T is bigger than cost of T prime or ((Refer Time: 26:27)) T prime has a smaller cost than T. But we had made a claim about T. So, we had claim the T is optimal. So, T is optimal for this space S i. So, T is the best possible tree of this form with I least on the left side it is a least cost tree. But, we have just found ((Refer Time: 26:52)), that t prime has even smaller cost than that. So, we have a contradiction.

So, we have a contradiction and therefore, our basic statement must be true. That L must be ((Refer Time: 27:07)) L must also be a least cost tree. So, if T is a least cost tree, then its left sub tree L must also be a least cost tree. But, ((Refer Time: 27:18)) we can apply the same argument to the right side as well. So, we can argue that the right sub tree of T must also be the optimal sub tree over it is leaves. So, we are now, ready to design a recursive algorithm.

(Refer Slide Time: 27:39)



So, we will call our recursive algorithm MCM stands for metrics chain multiplication. It will take as argument then array, the array which gives the dimension of the matrices. And it will return the optimal tree. We do not have the leaves the matrices themselves. So, the tree will just P the tree will just have an adequate number of leaves. And the leaves will serve as place holders for the metrics. We are only interested in the structure of the tree anyway.

So, we are not actually going to perform in the matrix multiplications themselves. But, we are just going to indicate the order in which the matrix multiplication should be performed. The basics for writing this algorithm is going to be the lemma that, which has proved. But we need to first take care of some base cases. So, suppose our array only represents a single matrix, in which case this n could be 1. We could be given just two number, the number of rows and the number of columns.

In this case of course, there is no multiplication to be done. But, this is the base case and here we will written a tree. But, this tree will just consist of a single vertex. Now, we are ready to generate the algorithm that we use the lemma generate our algorithm. The basic idea is that we are going to explore each subspace S_i . So, here is the code for doing that. So, I here is going to be the subspace that we are going to be exploring. And I will take value is 1 to n modulation one.

So, for each subspace we will first calculate, the best left sub tree that must be made that must form, the that must be a part of the optimal sub tree optimal tree for that. So, as for the previous lemma the left sub tree of the optimal sub tree optimal tree for this subspace must itself the an optimal tree. So, to do that, will recursively call MCM. But, this time we only want the optimal tree over the first I leaves. So, likewise we will only we passing the i plus 1 elements of this array.

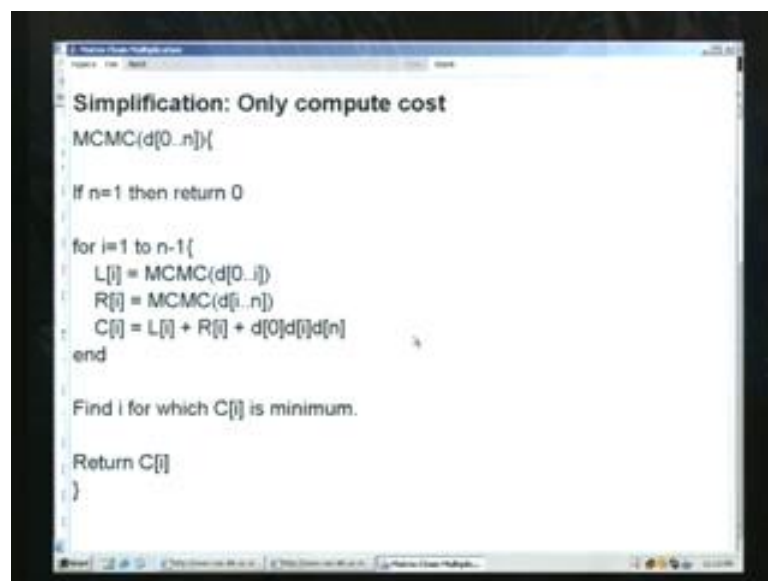
So, i plus 1 elements will define the rows and columns of the sizes of the number of rows and columns of the first i matrices and that is what we will pass. Likewise, will construct R_i , which will be the optimal right sub tree, which will be the part of the optimal tree for S_i . So, this will be done by this call MCM of d_i through n . And this will simply be the dimensions of the remaining n minus i metrics. $C_{sub\ i}$ is going to be the cost of this. This is going to be computed using this observation.

So, it is going to be the cost of L plus the cost of R ((Refer Time: 30:50)) and the cost of multiplying the two the results together. So, this is what we have over here. So, it is going to be cost of this tree, cost of this tree that we get and the time required or the cost required to multiply the results corresponding to these two. Note over here, that I am using this, this expression cost of L_i . And you may think of it either as a function, but more accurately you should think of it as a field selector.

So, when we return the tree itself, the tree will also contain its cost. So, we are just extracting the cost out of it. So in fact, as we designed the algorithm we should consider that we are not only written in the tree. But, we are also written in the cost of the tree. Next, we will have all these C_i 's and we just want to find the minimum C_i . And let us call that i for which C_i is minimum. And let us call that i , lets us use i to denote time.

And then, we are simply going to take the corresponding tree in the corresponding left and right sub trees and make a tree out of those and written that. So, this will typically we something like L_i will be a pointer to the left sub tree or R_i will be a pointer to the right sub tree. So, we will construct the new node, which will be the new root. And we will make the root point to align R_i . And since we also want to written a cost, we will attach an additional field, which is C_i over here. So, this is the end of the procedure. So, this is the end of the recursive algorithm that we wanted.

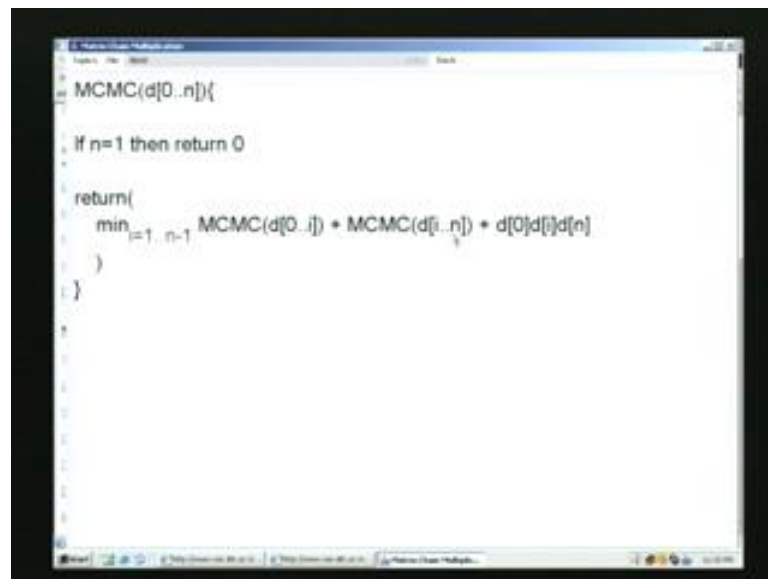
(Refer Slide Time: 32:39)



Now, we could work with this, but to simplify a matrix. We will only consider will consider an algorithm, in which we only compute we only compute the cost and not the tree itself. So, the idea is going to be very similar to the previous algorithm in that it going to be simpler. So, earlier we said that, if n is equal to 1 then be written single element. In this case, we are not going to written a element, but we are going to written its cost. So, in this case, we are going to written 0.

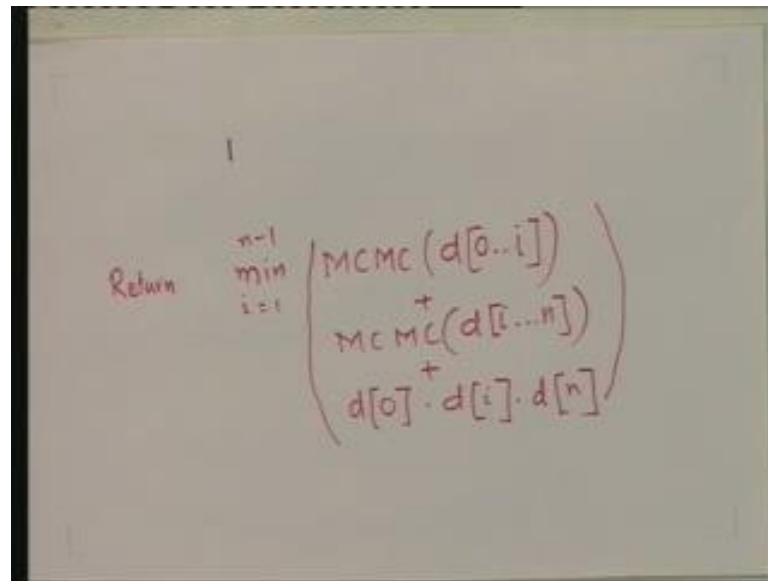
In the previous algorithm, we L_i equal to MCM of this. But here, now we are going to use MCMC. So, MCMC is the cost the cost of the optimal tree. So, will L_i will now become the cost, arrive will also become the cost. And so the total cost is going to be not cost of L_i , but just L_i plus R_i plus whatever this was. And now, we are going to find the i for which this is the minimum. And then we will write this C of i . So, we will work with this and the rest of the dynamic programming procedure will work will use this.

(Refer Slide Time: 33:51)



Actually, we would like to simplify, we can simplify the whole thing. And here is however, simplify it. So, let me go back to this and let me try doing this one paper ((Refer Time: 34:01)). So, you can see both the things at the same time. So, the idea over here was that we are going to calculate this expression and take the minimum. So. In fact, this whole part I can replace by the following expression.

(Refer Slide Time: 34:25)



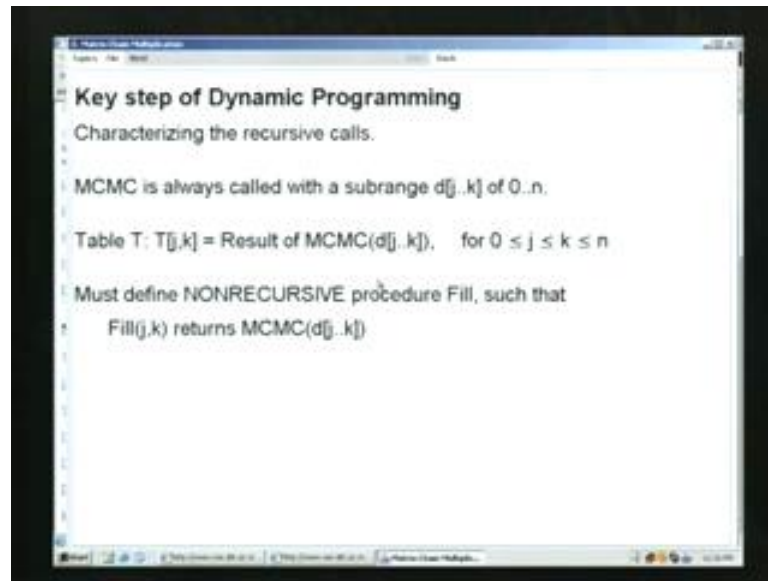
The image shows a handwritten mathematical expression on a whiteboard. The expression is written in red ink. It starts with the word "Return" followed by a large right curly parenthesis. Inside the parenthesis, there is a "min" symbol with "n-1" above it and "i=1" below it. To the right of the "min" symbol is a large left curly parenthesis. Inside this left parenthesis, there are three terms: "MCMC(d[0..i])", "+ MCMC(d[i..n])", and "+ d[0] * d[i] * d[n]". The entire expression is enclosed in a large right curly parenthesis.

$$\text{Return } \min_{i=1}^{n-1} \left(\text{MCMC}(d[0..i]) + \text{MCMC}(d[i..n]) + d[0] \cdot d[i] \cdot d[n] \right)$$

So, I will just write this as return the minimum over all possible values of i , of what are say in fact, i going from 1 to n minus 1. And what is it that we are going to take the minimum of it is just going to be this C of i over here. So, instead of L i , I will just substitute this MCMC. So, I will write that as MCMC of d 0 through i plus MCMC of d i through n plus d of 0 times d of i times d of n . So, I just taken this expression ((Refer Time: 35:25)) and I have substituted MCMC over it here.

And this MCMC over here and I get this expression. So, that entire part could be represented as this. So, we are going to take this expression for all values of i and we are going to find the minimum fit. And that is what we are going to return. So, this is the resulting expression ((Refer Time: 35:47)). So, if n is equal to 1 again will return 0 otherwise will return this, which is what I have return over here. The final step in dynamic programming is to characterize the recursive calls.

(Refer Slide Time: 36:01)



So, our recursive calls, which we made ((Refer Time: 36:04) for this procedure MCMC of d_0 through n . And we called d_0 over i over here and $d_i n$ over here. So, we MCMC ((Refer Time: 36:14)) was called these arguments. So, if we look at how MCMC would progress further, you will notice that each time we are taking our original range. And then we are splitting get in some way. So, here we are taking a prefix of the range. Here we are taking the suffix of the range.

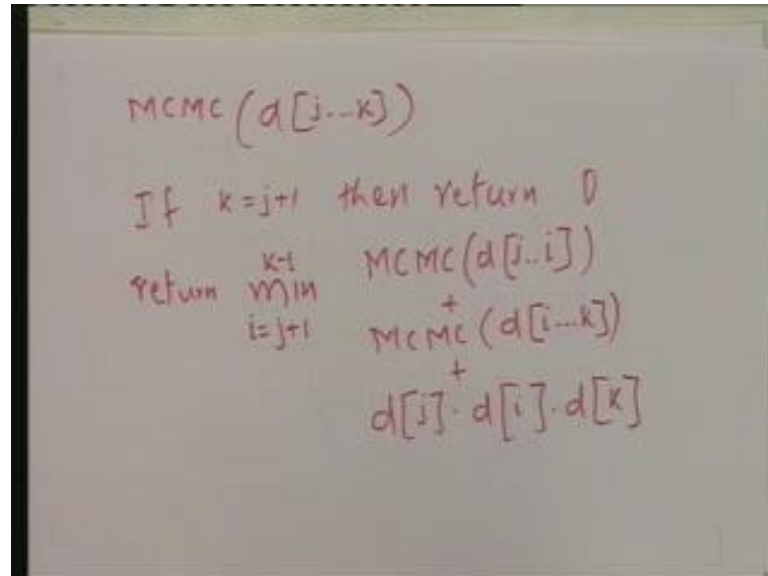
If we take a prefix of the suffix then we will get some range for not necessarily a suffix or prefix of this. But, never there is always some sub range of this. So, our characterization could be something like this. That MCMC is always called is a sub range $d_j k$, where $j k$ is the sub range of 0 through n . So, our entire array was d_0 through n . But, will call it with some $d_j k$. So, this allows just define a table.

Since, we know what all the recursive calls are going to be, we are going to allocate one entry for each possible result. So, here is the possible result. So, $d_j k$, where $0 \text{ less than, } j \text{ less than, } k \text{ less than } n$. So, this result of the call MCMC on this is going to be store in table entry $T_j k$. Our next step is to define a non recursive procedure fill, which will be used to fill this. So, if we call fill with $j k$, we should get exactly what is suppose to be filled over here or it should return MCMC of $d_j k$.

So, MCMC of $d_j k$, so you need to figure out what exactly is MCMC of $d_j k$. So, let us try and execute MCMC with the arguments $d_j k$. So, for that we need to go back to the

definition of MCMC. So, here is what we need to do, we want to execute the call MCMC of $d[j:k]$.

(Refer Slide Time: 38:09)

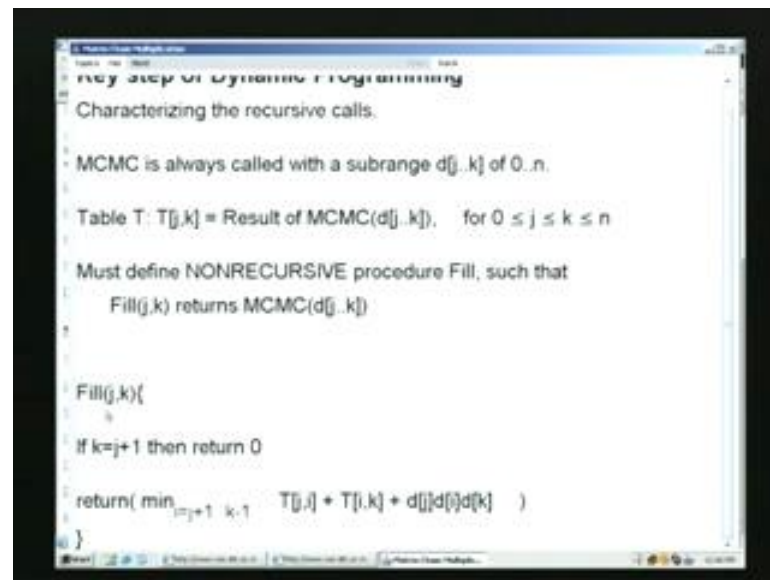


So, here is our basic procedure ((Refer Time: 38:25)) MCMC. So, this is being called the parameters over here are the entire sub range 0 through n. However, we want to call it with this sub range. So, assume of course, that the compiler will rename and make this appear like 0 through n. But, when it does that, what exactly will this return. What exactly how exactly will this execute. That is what we need to understand. So, the condition n equal to 1 over here. Just say, whether this last index is 1 plus this first index.

So, this condition really should be thought of us. If the last index k is equal to j plus 1. In that case return 0. Otherwise, it is going to return this expression. So, we are still going to return this expression, but wherever the first index 0 appears will just put in j and wherever the last index n appears we just put in a k . So, that exactly what we are going to what that procedure should be doing. So, MCMC even called on $d[j:k]$ will work something like this. So, it will return say min our i going from not 1 to n minus 1.

But, instead it will go from j plus 1 to k minus 1. And it will compute MCMC of d of j i plus MCMC of d of i k plus instead of this we are going to have the corresponding right bounds ((Refer Time: 40:29)). So, we are going to get d of instead of 0, we will get j times d of i times d of k .

(Refer Slide Time: 40:45)



So, this is the value that should be return by fill. So, here is what fill in fact, as. So, fill is going to do exactly this ((Refer Time: 40:56)). However, fill is not really allow to reference MCMC and in fact, filled does not need to reference MCMC. So, when it wants to compute MCMC of $d[j..k]$. It fill knows that this is going to be stored in $d[j..k]$. So in fact, instead of MCMC fill is just going to return $d[j..k]$. Instead of MCMC of $d[j..k]$ fill is just going to return $T[j,k]$. And then this part is going to be the same as before. In fact, that is what fill is going to be returning.

So, this procedure fill that we have defined over here is the right procedure it is non recursive. It does not think talk about MCMC does not call anyone else, but it is look at the table entries. So, this is what we needed, we want it to have a procedure, which tells us how to fill the $j..k$ th entry assuming that other entries are full. So, this is what it is doing. So, now we are really pretty match ready to right the dynamic programming algorithm. So, what we need to do next is to characterize how fill works. So, let me write down over table first.

(Refer Slide Time: 42:22)

[illegible]

So, our table T is going to look something like this. So, here is the first index, index which is i here is the first index, which is the second index which is j, sorry our first index is j and the second index is k. So, let us now look at the first step the first statement and fill. So, it is says that fill ((Refer Time: 42:56)) the table must contain as 0, if k equals j plus 1. So, this is 1 1. So, this will be the entry 1 1. So, this is not of interest. So, if k is equal to j plus 1. So, this entry is going to be a 0. This is where k equals j plus 1.

But, that is not the only entry. In fact, another entry, which will be 0. Because, of this will be this entry and so on. So, these are simply entries above the mean diagonal. So, this could be the mean diagonal. And all the entries above the mean diagonal will be 0's. And in fact, the whole thing is going to be define for j less than or equal to k . So, in fact it is going to be defined on this side. This side is not going to be use, this side and the diagonal are not going to be used at all.

Now, let us examine how does fill ((Refer Time: 44:01)) a particular entry j k . So, let us say this is my j over here. And so this is the entry that I want to fill. And this is the index k . So, the j th row k th column this is the entry that I want to fill. How does filled ((Refer Time: 44:21)) fill this entry. So, it considers the values of T_{ji} , where j ranges sorry T_{ji} where i ranges from j plus 1 to k . So, these are simply table entries in this region. So, let me put them in a different color, so it so these are the table entries.

So, these are the table entries, which are needed for updating this. So, let me put an arrow going from here to here. But, it also uses table entries i k ((Refer Time: 44:53)) again I going from j plus 1 to k minus 1. So, which are these entries. Well, these are the entries, which are below over here. So, the entry is above the main diagonal. But, below this element are also going to be used. So, if these entries have been have already been defined. And these entries have already been defined.

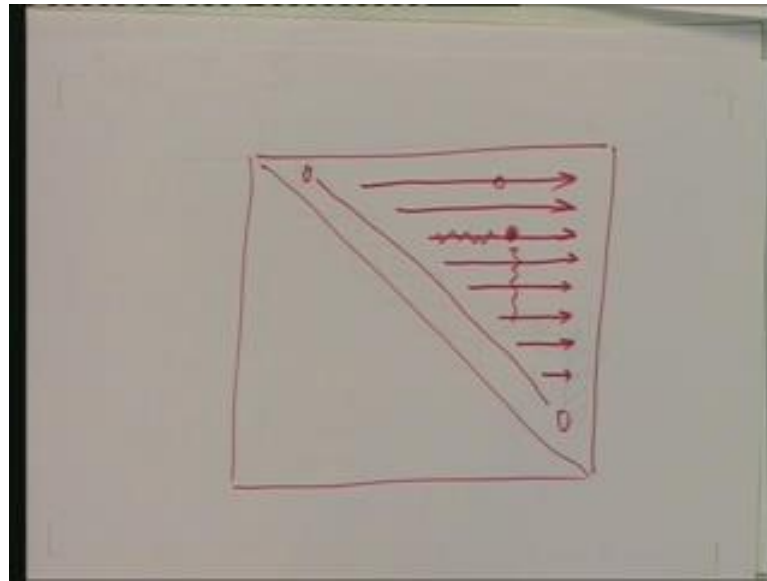
Then fill can fill in this table entry. So, then the entire procedure is extremely simple. What is that we have to do. We must ensure that when it comes time to fill this entry. All these entries must already have been filled. So, ((Refer Time: 45:41)) that such as is a very simple order in which we can fill these entries.

((Refer Slide Time: 45:47))

7	8	9	10
0	4	5	6
0	2	3	
0	1		
0			

So, here is my table. So, here is the main diagonal. So, this diagonal I am not going to do anything with. But, the entry is above that I am going to mark as 0's. Then, I am going to start filling entries. But, I should fill entries such that if I want to fill this entry. Then, everything on this side and everything on this side is already filled. So, next I can fill this entry maybe. So, I will put a 1 over here. After that, I can fill this entry. I can fill this entry. Then I can fill this entry, this entry, this entry, 7, 8, 9, 10. In general, I can let me use another picture. I can take my matrix.

(Refer Slide Time: 46:39)



So, this is my matrix, this is my table. I will filled the second diagonal with 0's. Then, I will fill the entries, the rest of the entries in this order. First, I will fill these, then I will fill these, then I will fill these, then I will fill these and so on. The point is that, if I have fill entries in this order. Whenever, it comes time to fill say an intersome where over here. I would already have filled these entries and these entries on which this entry depends. So, that is the key idea. So, if these entries are already filled then you can fill this entries. So, the only thing that remains to be done now is to figure out, how much time the whole thing ((Refer Time: 47:28)) will take.

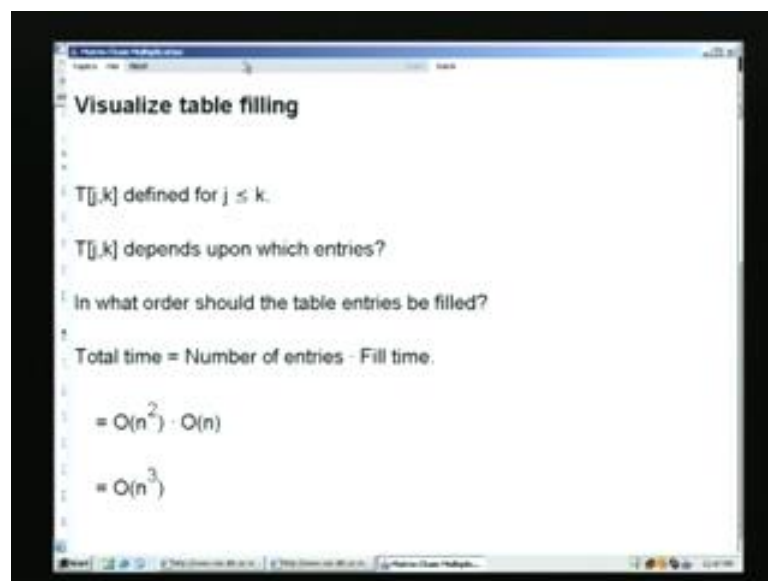
(Refer Slide Time: 47:32)

Total time
= number of entries in
table \cdot Time to fill
1 entry.

So, the total time taken is clearly equal to number of entries in the table. Times time to fill 1 entry. So, this is already telling us, everything that we want to know. How many entries do you have in the table. Well, T has all this entire range $j \leq k$, where j, k lie between 0 and n . So, we have a two dimensional table with n rows and n columns. We are only using half of it. But, clearly the number of entries is therefore, O of n squared. What about the time to fill a single entry.

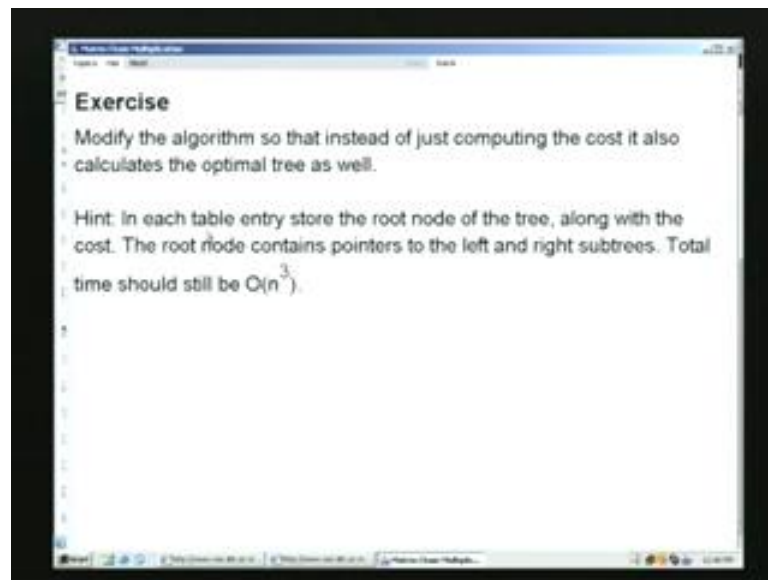
So, $T[j, i]$ and $T[i, k]$ are already known, when we ((Refer Time: 48:29)) are going to have to compute this product. And we are going to do these additions. But, and this has to be done these many times, so far every value between j plus 1 and k minus 1. This range can be as large as n . And therefore, the time to fill a single entry is going to be O of n again. So, the whole thing is going to be O of n cube. So, that really ((Refer Time: 48:58)) finishes the entire algorithm.

(Refer Slide Time: 49:02)



So, the total time is n square times n which is equal to O of n cube. Now, I just want to make one more comment. In the comment is about this simplification that we tell. So, we said that, we are not going to compute the exact tree that we wanted. But, we are just going to compute, the cost of the optimal tree. So, I want to leave you with an exercise, which is how can you modify this algorithm. So, that you actually, calculate the tree and not just the cost.

(Refer Slide Time: 49:44)



So, this is exercise. So in fact, the idea is something like this. So, I am giving your hint over here. In each table entry, store the root node of the tree along with the cost of that. The root node will contains pointers to the left and right sub trees. Now, if you go and ask the question, how do I update a single element of the table. You will have to construct the root node as well as these pointers and as well as these cost. But, you will see that each filling now will still be possible in O of n time. And since the number of entries are O n squared the total time should still be O of n cube. So, that concludes this lecture.