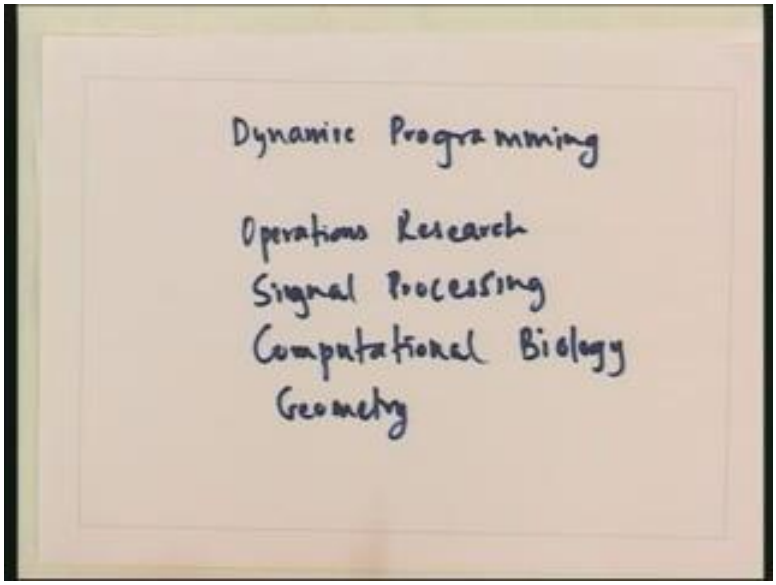
Design and Analysis of Algorithms
Prof. Abhiram Ranade
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture - 18
Dynamic Programming

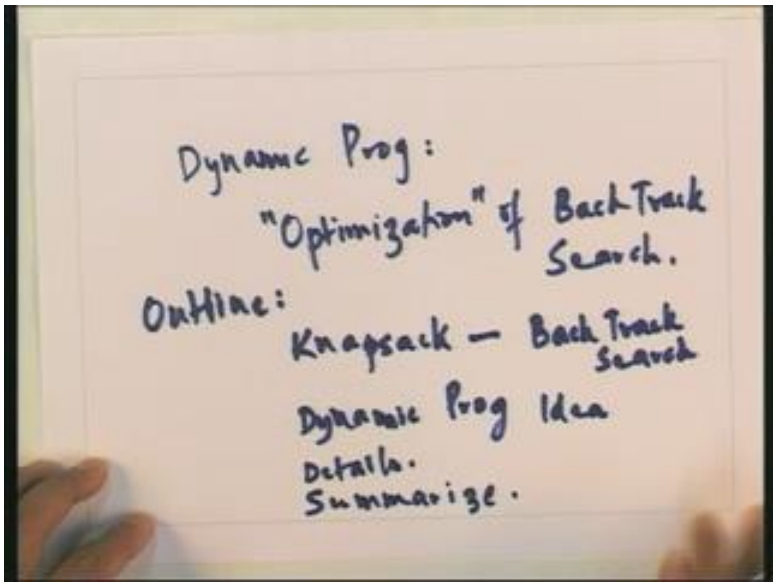
Welcome to the course on Design and Analysis of Algorithms. Our topic today is Dynamic Programming. Dynamic programming is a powerful technique for designing algorithms, and it actually finds applications in many areas.

(Refer Slide Time: 01:10)



The name itself is not really very, very enlightening. So, I will not talk about it, there is an origin is somewhat obscure. And so let us not worry about it. But, as I said the techniques finds applications in many areas, including operations research, signal processing, computational biology, geometry and many more. We are going to view, dynamic programming as a technique for combinatorial optimization. And in some sense, will think of dynamic programming as an optimization of the backtrack search technique that we have seen so far.

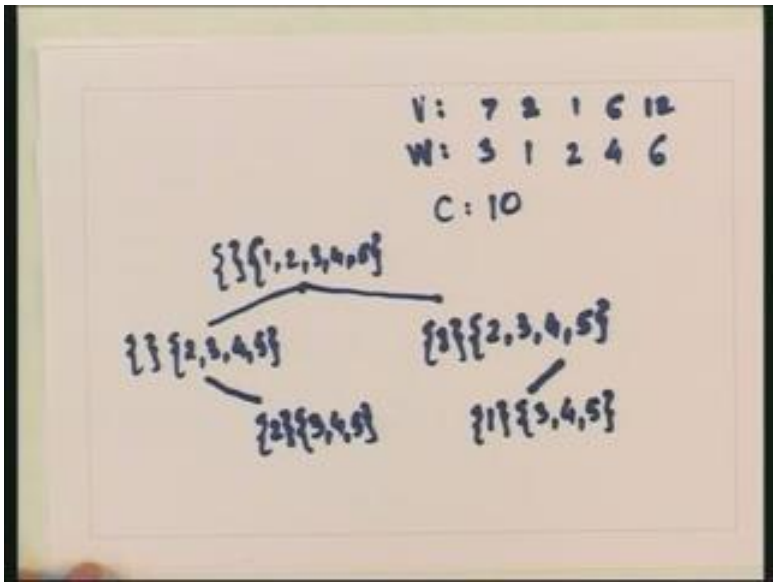
(Refer Slide Time: 02:29)



So, we will think of dynamic programming, as in some sense are optimization of backtrack search. So, here is what I am going to do today. I am going to talk about, I am going to take an example to illustrate the technique and this example is going to be our familiar knapsack problem. So, I am going to review the backtrack search solution, then I am going to say how we can really ((Refer Time: 03:03)) slightly differently and how we can optimize it. And that is how we will lead into the dynamic programming idea.

And then I will describe some details and then summarize. So, let me take an example of an knapsack problem. So, let us say our knapsack problem, let me remind you that our knapsack problem involves filling a knapsack with objects of the maximum value. The inputs to this problem are two vectors.

(Refer Slide Time: 03:55)



Let us say V, V is one of the vectors, which gives the value of each object. So, let us say we have V value is which are 7 2 1 6and 12. This juts mean that the first object has value 7, the second object has value 2, the third object has value 1, fourth object is value 6 and so on. Second thing of the value has been given to us a rupees or something like that, then we are also given another vector. So, this vector gives the weight of each object.

So, for example, we might have that the first object weights 3 kilograms, the second object based 1 kilogram, the third object weights 2 kilograms the fourth object weights 4 kilograms and say the last object is 6 kilograms. And finally, we are given a knapsack and it is capacity. So, this is our last parameter C and say this time C is given to be 10. So, let me remind you what the problem is, we are suppose to pick up objects from the set, such that the total weight is at most 10.

But, we want to pick up the objects of maximum value. So, this is the problem. Let me now remind you, how we did backtrack search on this. So, we become the backtrack search, by saying that at present, we have all possible objects in front of this and we have not picked up any object. So, that is corresponds to the first node in our search. So, let me draw that node out here.

So, this first brace says that we have not really made any decision yet, on any object. We have not picked up any object yet and the second set 1 2 3 4 5 gives us the list or this is the set of objects, which I still need to make a decision about. So, this first vertex just

says that I have to make a decision about all objects. And backtrack search, systematically goes through the entire search space and determines the value, when different objects are picked or not picked.

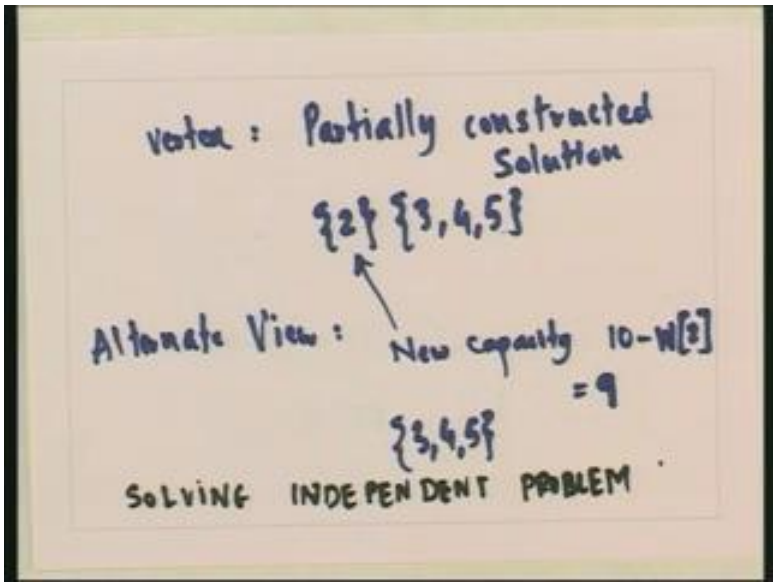
The first decision point is where we decide, whether or not to pick up the first object. On this side, I am going to consider the case that the first object, in fact is not picked. So, for example, I will write here, nothing has been picked yet. But to indicate the fact that I have made a decision about the first object, I will remove the first object from the set of objects, which about which I have to make a decision. So, I will just write 2 3 4 5 over here.

On this side, I will make the other choice which is, I will pick up 1 and then on this side I will include 2 3 4 5. This means, that I have decided to pick up the first object and these are the objects about which, ith have to make a decision. In a similar manner, I will keep on examining each of these nodes, I will make more and more decisions. And I will systematically evaluate all the possibilities.

So, for example, on this side I could have said that, so next decision is about object 2. So, maybe I will on this side, I will not pick up object 2 and this side maybe I will pick up object 2. So, now the set of object that remains unpicked is 3 4 5 or the set of object, which remains undecided is 3 4 5. On this side, similarly let us say I go to the left and as I have said we have by convention decided that on the left side, I will not picking up the corresponding objects.

So, again the objects that I have selected are only one. Whereas, I just made a decision about object 2 and I said, I am not going to pick it up. So, the objects that are remaining are 3 4 and 5 and so on. I will explore the tree in this manner and we remark long ago that if we do this, we will generate a tree with height with total number of leaves 2 to the n. If n denotes the number of objects that clearly is two large at time for even for moderately large n. So, we clearly need a better algorithm than this. Now, I am going to take a different view of this backtrack search procedure that I just described. Let me first remind you, what the backtrack search view is…

(Refer Slide Time: 08:53)



So, the backtrack search view of a vertex in this tree, corresponds to a partially constructed solution. So, for example, I wrote 2 and 3, 4, 5, this just says that I have included 2 in a solution. I have decided not to include object 1 and this is the part about which, I have not yet made any decisions. But, here is an alternate view, suppose I have decided to include 2. So, let me go back to my old problem.

So, if I decide to include 2 and not 1 ((Refer Time: 09:50)) and I have capacity 10 then; that means, that out of the capacity that I had of 10, I have used up capacity equal to 1. Because the second object has weight 1. So that means, corresponding to this vertex. So, this choice of mine really says that our new capacity after I have decided to include object 2 is 10 minus weight of the second object. In this case, 10 minus the weight of second object happens to be 1.

And therefore, the new capacity happens to be 9. And then, I have not yet made decisions about objects 3, 4, 5. So, rather than thinking about this remaining part, as an extension of the old solution, this new view encourages us to think. Let us forget our original capacity. Let us say deal with what really is remaining. What really is remaining is 9. So, we have capacity of 9, which is remaining. And we have to select the 3, 4, 5, these objects 3, 4, 5.

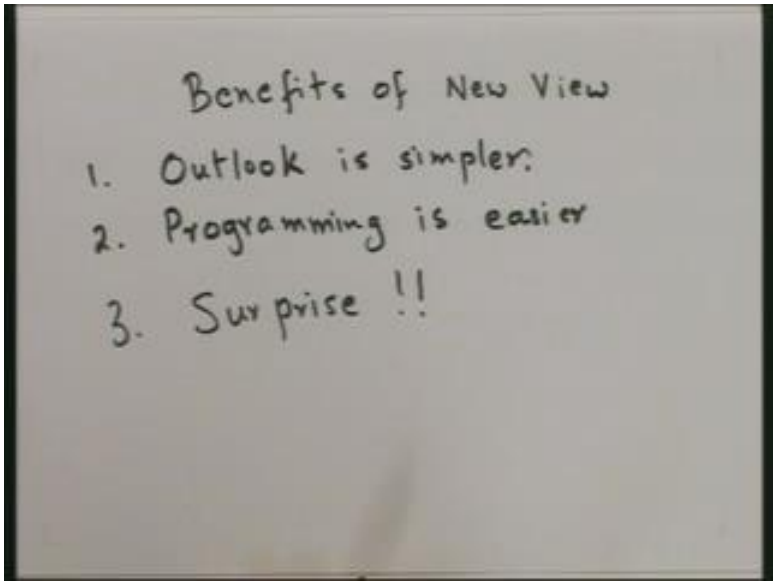
So, we are allowed, now to take objects from this set 3, 4, 5. Our capacity is 9; that means, the weight of objects, which is select from this must not go beyond 9 and within

that, we want to maximize our value. Clearly solving this problem, extending the solution over here is exactly the same as this, as solving this independent problem. That is because even here, I will have used up 1 unit of capacity, out of the 10 units that I have already had.

And clearly, it makes sense for meet to fill up the remaining capacity with the best, the most valuable objects from this set. Whose total weight is less than 9. And that is exactly what happens here as well. So, this is; however, a substantial improvement because, now our view is that, we are not really thinking of about extending the solution. But we are thinking of solving and independent problem. And further that problem is of the same kind as the original problem.

So, rather than saying that let us extend the solution, we will just say that look in order solves this original problem. Let us solve this alternate problem, but this problem is not an extension, of any extension of any solution. But, it is a simple new problem of the same kind. So, this first of all introduce as simplification in our outlook. And in some sense it makes programming easier, so the benefits of this new view.

(Refer Slide Time: 13:15)

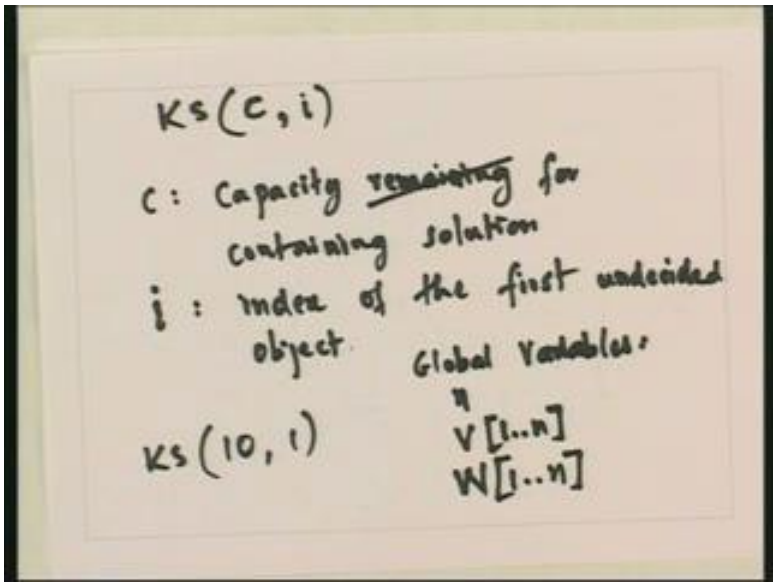


The first benefit is that, this new view is simpler or I would state that as the outlook, our outlook overall is simpler. This in term such as our programming is going to be easier. I will talk more about this in a minute and you will see that indeed, the programming gets simplified. However these two are not really, the main important benefits. There is an

additional benefit, which is much more important, but about which I am not going to tell you right now.

Let us wait, until I tell you of how to do the programming and then I will tell you, about this surprise benefit. And you will agree with me then, that this is really a very exiting benefit. So, let us start, let us worry about, how we going to program this. So, the idea was that in order to solve the knapsack problem. Instead of thinking about extending the solution, we said let us construct a new problem of the same kind and then solve that maybe, one or as you have seen maybe, more problems of the same kind and solve that. So, this naturally means that we should be thinking of recursion. So, let me write down a recursive procedure, to solve this knapsack problem. So, let me first define that procedure, let me write down this specifications of the procedure first.

(Refer Slide Time: 15:18)



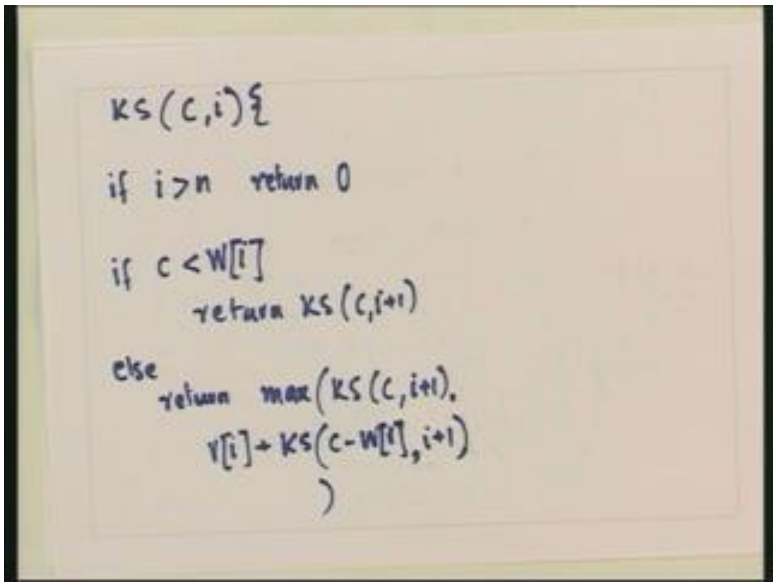
So, I will call my procedure Ks and it will take two parameters C and i. C is going to be the capacity, remaining or actually I should say rather, than remaining that the capacity given for solving the problem. So, let me just strike this off, capacity for containing solution. I is a parameter, which is the index of the first undecided object or in fact, instead of thinking about the first thinking of it as a first undecided object, I could think of it as the first object, which is allowed.

So, the object which are allowed for me ri, i plus 1, i plus 2 all the way till n. So, my original problem that I needed to solve was say something like Ks of 10, 1. What does it

indicate that I want to solve the knapsack problem, on knapsack of capacity 10 and I want all the objects, in the first object onward until n to be considered. I have told you what n is and let us assume that n is a global variable. So, there are some global variables over here.

So, n is a global variable, the total number of objects, similarly the value array is a global variable and the weight array is also a global variable. And let me just remind you, that there are n components in each of these arrays. So, these are the specifications of the procedure that I am going to write. So, let me now write the procedure. So, let us deal with the base case first that is fairly easy.

(Refer Slide Time: 17:49)



Ks of C i, so first the base case, so if i is greater than n. So, that means that I have been asked to solve the problem, starting from the n plus first object or some object beyond n. That means I really have made up my mind about all the previous object ((Refer Time: 18:19)) or in essentially what is am saying is that, I am not allow to choose any objects at all. In that case we should be return in 0, as the value of the solution. I should say before I get into this, that in this problem that this specification that I have wrote down. ((Refer Time: 18:42))

I should also say that we will only we worried about the value of the optimal solution. So, the idea is that case of 10, 1 will not actually return, the set of object which are chosen. But, in fact it will return the best possible value that I can get out of it.

Now, (Refer Time: 19:14) I think this is a (Refer Time: 19:15) simplification of the problem that is given to us. And in some sense it is, but later on you will see that. In fact, once we know what the value is, it is very easy to go back over our calculations and actually figure out what that set was. So, we will leave that for the future, later in this lecture. Right now we will just concentrate on determining, what is the maximum value I can get by filling my knapsack, so that it does not overflow, it is in overloaded.

This is the base case, if I have already gone past n then; that means, we have considered all the possible objects and then we should returning 0. Because there is nothing to add to our knapsack, so you have already return. Otherwise, the first object that we are going to be considering is the ith object. So, what can you do about it, well we need to consider it only if the capacity is in fact, able to take that object.

So, if C, if the capacity happens to be less than the weight of the object, then we cannot do anything right. So, if the capacity happens to be is less than the weight of the object. Then that object cannot be considered, which means, we should just go on to the next object and return whatever we can with the remaining object. So, in that case, we are going to return Ks of C and i plus 1.

The interesting case appears, if C is in fact, bigger than W of i, so else what happens. So, else we have the possibility that we can in fact, include this object in our knapsack of course, it might not always be wise to include that object. Simply because, if we include that object we have to reduce, we have to use of some of your capacity and maybe the benefit that we get because of using of that capacity is not adequate enough. Or because we might be losing some better options later on.

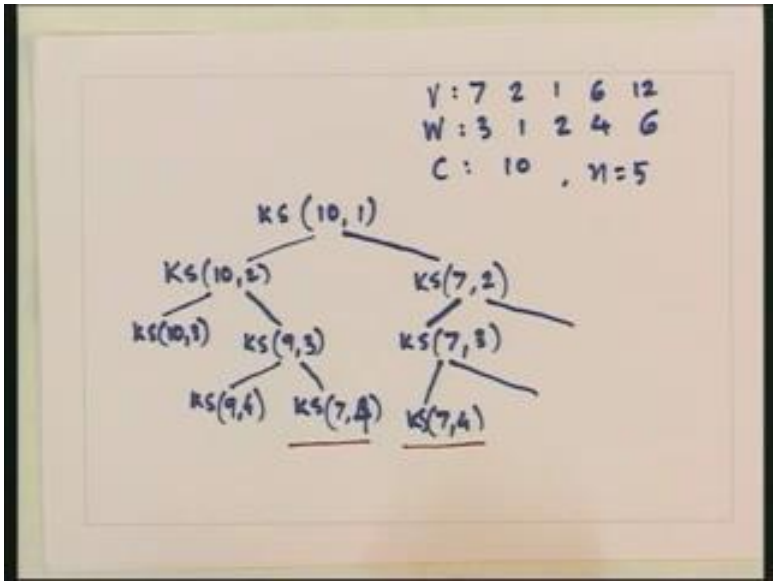
So, therefore, to cover both possibilities, here is what we should do. We should return the best of those towards. So, the first is we will ignore this current object, in which case we should be returning Ks of C, i plus 1. And then we need to consider the other possibility, which is that we include this ith object. But, if we do include the ith object there are two effects.

First of we gets it value. So, we should be returning in whatever we return, we should improve that value as well. So, we are going to return V of i, but will also get some value form the remaining objects. However, when we go for the remaining objects we would

not have the full capacity C to consider. But, instead will have a capacity, which is a little bit reduced.

So, we should we should be returning, V i plus Ks of C minus width of i, because this is the amount by which the capacity series is going to be reduced. And whatever choice we can make from i plus 1 from the remaining objects. So, this is what we should return, the best of this quantity and this quantity. So, this finishes the recursive implementation of the backtrack search that I have just mention. Just make sure, that we understand this. Let us actually execute this, on a sample on our own example, in fact.

(Refer Slide Time: 23:49)



So, let me write down what our Vs and Ws are. So, our Vs and Ws were V was 7 2 1 6 12. W was 3 1 2 4 6 and our capacity was 10. We started of by making the first call, which was Ks of 10, 1. Ks of 10, 1, now let us go back to our code and our value of n was equal to 5. So, if we go back to our code and ask how this code could execute, the first step we have to check is i bigger than n. n is 5 and i is 1. So, clearly it does not happen and. So, we come to situation.

So, the weight of the first object is 3. (Refer Time: 24:46) And the capacity is certainly bigger than that. So, this is not case that we execute, but in state we execute this case. In that case, we are going to consider two possibilities. One is searching the knapsack possibilities, with the same capacity and i plus 1. So, let me draw this as a recursion tree.

So, here I am going to get Ks of 10, 2 and of this side I am going to get whatever is whatever corresponds to this. ((Refer Time: 25:19)) But, corresponds to this we are going to use up, out of the C capacity we are going to use a W i capacity. So, W of 1 capacity is going to get used up the W of 1 capacity is 3 units. So, on this side I am to I am going to make a call Ks with the reaming capacity, which is 7, 2. Because, now I am going to only consider objects starting at the second object.

In this manner, we are going to execute. So, let me do a few an executions, just to make sure that the idea is understood. What happens, when we try Ks of 10, 2 again even here, we will see that ((Refer Time: 25:57)) the weight of the second object is 1 and that is still smaller than the capacity. And therefore, we will use this part of the statement. So, the first thing will execute is Ks of C, i plus 1, which is with the same capacity will try to go for the remaining object.

So, will be executing Ks of 10, 3 and on this side, we will be executing recursive call corresponding to this call over here. ((Refer Time: 26:33)) W of 2 now is this. So, will have to decrement that 1 from that and therefore, the call that we execute over here will be Ks of 9, 3. Let us do one more of these calls. So, let us see what happens when case of 9, 3 is executed further. So, again if you go back to this procedure, you will see that even here the capacity is larger than this W I, W of 3.

W of 3 is 2 and it is still larger and therefore, this will get executed.(Refer Time: 27:09) So, again there will be two children, two recursive calls. So, the first one will simply be Ks of 9, 4 and on this side, the call would be Ks of C minus W i. So, C minus W i this time is this. So, it is going to be 7, 4. Let us do something on this side and. So, what happens over here. So, case of 7, 2 if we execute we are going to get something like this.

So, first we will execute Ks of 7, 3 this corresponds to the possibility that we do not choose the third object. They will of course, be corresponding exploration on this side where, we do choose the third object. But, even here there will be two possibilities Ks of 7, 4 and on this side the possibility that we choose the third object. So, whatever that is. At this point I would like you to do two things, first of all I like to make sure that this picture is understood.

What we have drawn in this picture is, what is popularly called recursion tree. So, this is the first recursive called we need, that call give raise to this recursive call and this

recursive call, that in terms give raise to these two recursive calls just will give raise to this call and some other call over here and so on. But that is of course, quite routine, the most interesting parts of this picture are these two calls.

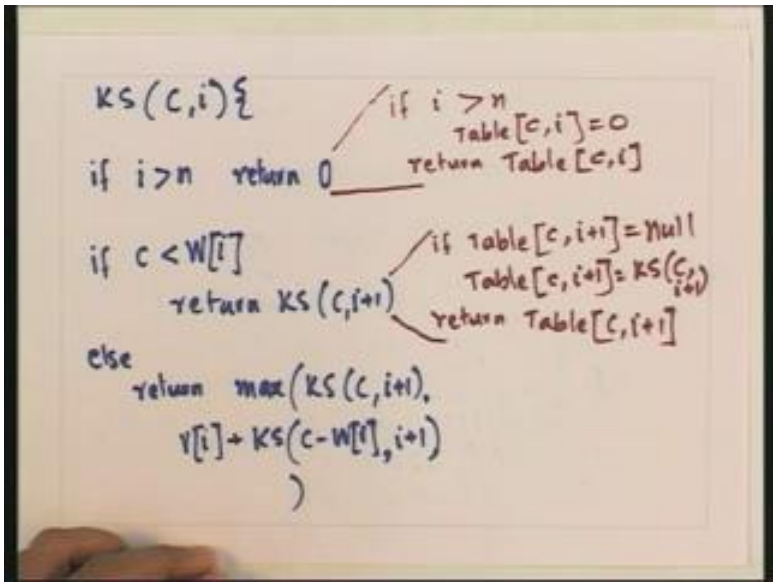
This should really write as Ks. So, there is something very interesting about these two calls. They are identical. What is it mean, in this part of the tree I am going to make a procedure call with parameter 7 and 4. And I am also going to make procedure call this side, also with parameters 7 and 4. So, this is the beauty. So, this is where the optimization can come in.

Once I explore this search tree underneath this, I won not need to explore this search tree again. If I store the value, that I get from this call, then when I come to this portion. If I just need to look it up, I will just look it up and I will get it. In fact, that is going to be my default idea. Whenever I calculate the value of a certain recursive call, I will actually stored it in a table.

Before embarking on any recursive call, I will first check if the table already contains are value, if it does I will just use that value. If it does not then I will calculate that value. But at the end of it I will stored it in the table that is basically the idea that is basically the optimization that I was talking about. And you can see that by viewing, what remains to be done rather than thinking of this search tree as an exploration, in which you are extending solutions.

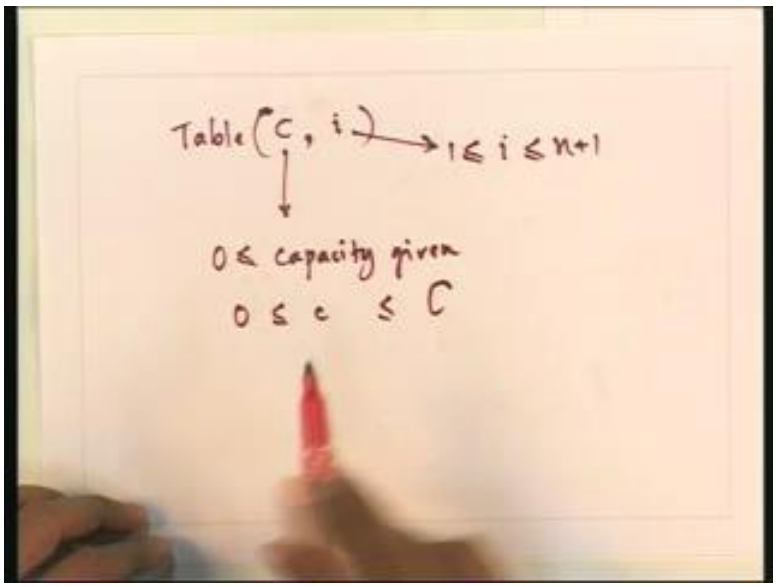
It is in this new view of things; it is possible to determine that the work over here is the same as the work over here. And thereby, we can do this optimization that we that I just mention. So, let me let me now flush out this optimization for you. So, we will go back to the same code and now I will write down, what is the corresponding code with the optimization. So, let me write that down in different color. So, if i greater than n then return 0 is the idea over here. So, we said that before returning any value, we are going to store it in some table.

(Refer Slide Time: 31:29)



So, instead of that instead of just returning 0, this part of the code will be replaced by this part. So, we are going to still check, if i is greater than n. But if it is then we are going to we are going to set table of C, i equal to 0 and then we are going to return table of C, i. So, this is what, this statement is going to be replaced by in our optimized version. Let me just remind you that, that the idea is to remember value is before we calculate them.

(Refer Slide Time: 32:21)



So, we are going to have another global array called table. And table is going to have, this is going to be a two dimensional array. So, we are going to have, the first index

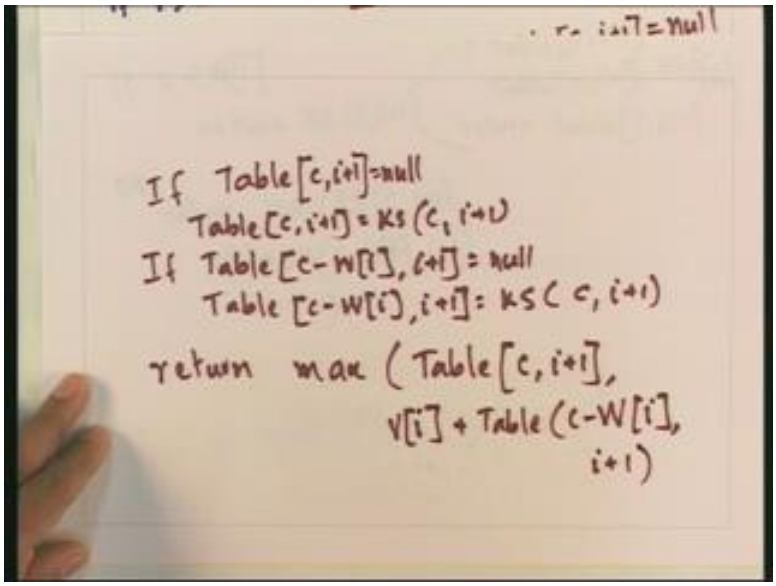
correspond to all possible capacities and the second index correspond to all i's, all the possible values of i. So, this is going to be, i is going to be from 1 to well n and as we saw our base case actually takes at beyond n. So, it is going to be n plus 1. So, the second dimension is going to be n plus 1, C is going to be in the range 0 through whatever value, so the capacity given.

So, in fact, let us say this is little c and. So, little C 0 is less than or equal to little c, less than or equal to capital C. So, the first index of the first dimension will have this range, this is the two dimensional array in which, we are going to store our values. What about this if C is less than W i return Ks of C i plus1. We just define the rule and we are just going to follow it. So, instead of returning case of C i plus 1, we are going to first check. So, the way we check, I will just write down the code corresponding to this.

So, we are going to check, if table of C, i plus 1 is not calculated and that I will denote by null. Then we are going to actually calculated. So, will calculate table of C, i plus 1 is equal to Ks of C, i plus 1. And then we will return table of C, i plus 1. So, this is going to be, what this code is will replace this code over here.

Similarly, we will have code, which will replace this as well. So, corresponding to this we will just make a check in C i plus 1 here. So, this part will really with the same as this and instead of this, will make a check in a slightly different position. So, let me write that down as well. So, let me write that down separately over here. So, return ((Refer Time: 35:19)) of C i plus 1 is going to be done by first checking whether case of C i plus 1 has been calculated.

(Refer Slide Time: 35:28)

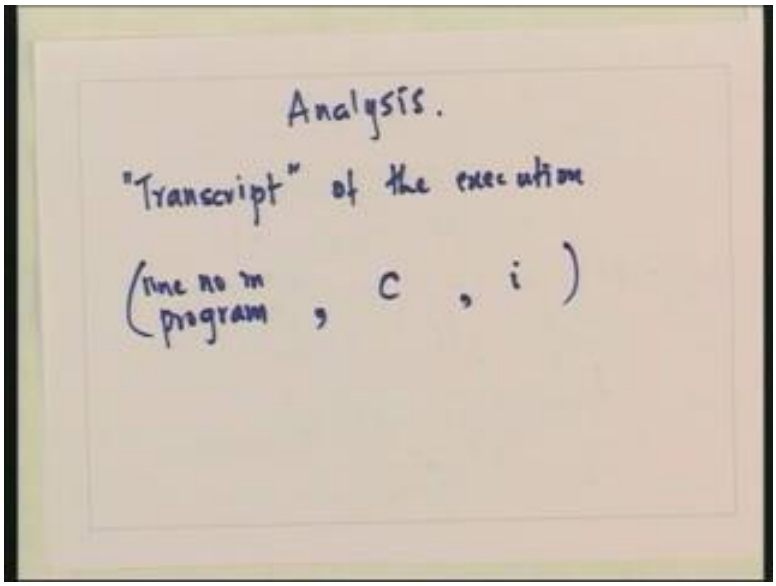


So, that is as good as saying if table of. So, return this expression is going to be done by checking this. So, we are going to check if table of C i plus 1 is equal to null. So, in that case we are going to calculated. So, we are going to set table of C i plus 1 equal to Ks of C i plus 1. Then we are going to check whether this has been calculated as well, ((Refer Time: 36:15))

So, that is as could as saying, if table of C minus W i, i plus 1 is equal to null. Then we actually calculated, table of C minus W i, i plus 1 is equal to Ks of C i plus 1. And then finally, we will just return the maximum of these two quantities ((Refer Time: 36:54)) and that is done simply by saying, return max of table of C, i plus 1.And v of i plus table of C minus W i, i plus 1. So, this code will replace this last code in your program.

So, should be clear to you, that this new code that we have return. Because, it is going to do less work because some parts of the search tree, which were explored several times in the original code will now be explored exactly once. But, now you would like to prove by how much precisely the work gets reduced.

(Refer Slide Time: 37:56)



So, we come to the analysis. Normally when you write recursive algorithms, the idea is that will write to recurrence for the time taken and we will saw (Refer Time: 38: 07). In the case of dynamic programming and in particular, where we stored these values in a table and use them as needed, this recursive estimate is not going to be very good. It is going to be an over estimate. So, we need something much more precise, we can produce an estimate; we can produce a way of doing that estimate, which gives us much sharper bounds.

So, here is the idea. So, think of this way as the computer executes each line of the code. Suppose, it writes the diary, in the diary it is going to write the following things. So, I am going to call this diary, it is customary to call this diary a transcript. So, this is going to be a transcript of the execution. So, the computer is going to write down, the line number in the program. Then it is going to write down, the value of C that is the correctly it has and then the value of i that it currently it has…

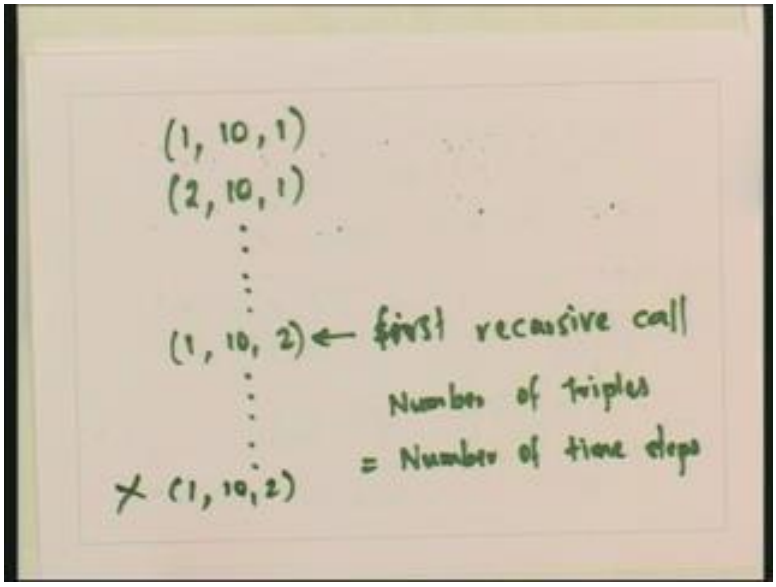
It is going to write down a triple, like this. So, it will write down once such triple every time it executes a line in the program, where we did in this. This will become clear in just a minute, if the idea is clear. So, let me take an example.

So, here is my code. So, let us say we have number the lines in this code, some going to for simplicity. I am going to think of this has been my code or actually I could do this as well. ((Refer Time: 40:06)) So, this is my this is line number 1, this is line number 2, this

is line number 2 and so on. So, give numbers to every line in this program. And I have added some lines over here. So, I will give numbers these lines as well, maybe this is line number 14, (Refer Time: 40:22) this is line number 15, this is line number 16 and so on.

So, I have given numbers to everything whatever, this is line number 20. So, when the programs starts executing, I am going to be making the call Ks of C i. So, what will the computer write the first time around. So, this is going to be my line number 1 whatever, this is going to be my number 1, this is going to be line number 2 and so on.

(Refer Slide Time: 40:54)



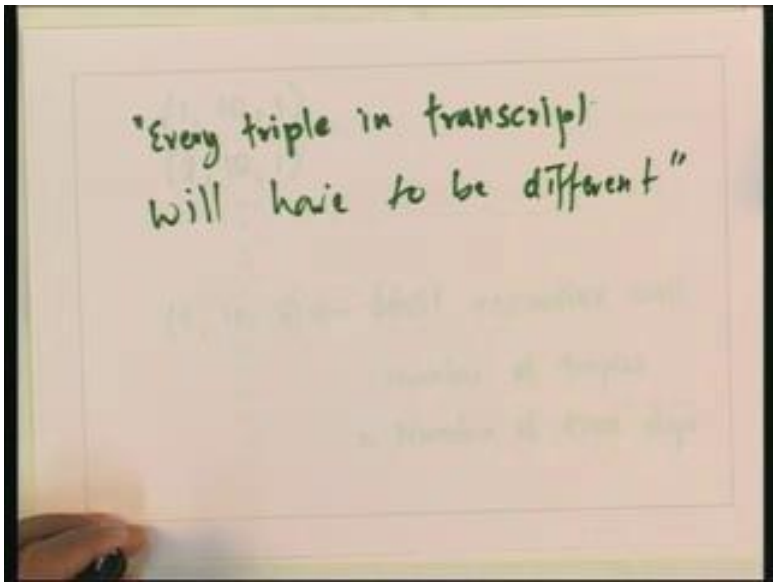
So, my transcript of my program is going to be the line number, which is 1, what will be the value of CB, the first time the value of C is 10 and the value of i is 1. So, this will be the first entry in my transcript. The next entry in my transcript is, I am going to execute the next line of the program 2, 10 and 1. As I keep writing at some points, things will change, of course the line number will change almost every time. But, at some points this 10 will also change, the 10 will change according to my execution tree, which I drawn over here.

So, in this execution tree as I started up with Ks 10 1, but then I went on to Ks 10 2. So, at some point during this execution, I am going to get again the 1, 10, 2. So, this corresponds to the first recursive call. And similarly, the different recursive calls will come out over here. So, this is going to be my transcript. How will you lines will there be

in this transcript. How many triples will there be in this transcript, clearly number of triples is exactly equal to the number of time steps.

Because at every step, the every step the computer takes, it is going to make one entry into its diary or it is transcript and that is going to be triples. So, the number of triples is exactly equal to the number of time steps. So, if I want to estimate the time taken by this computer on this particular problem, all I need to do is to count how many triples, I have in my diary or in this transcript. You might now wonder, what is the big deal we want to estimate time, now we want to estimate triples there is actually rather interesting property to do with these triples.

(Refer Slide Time: 43:08)



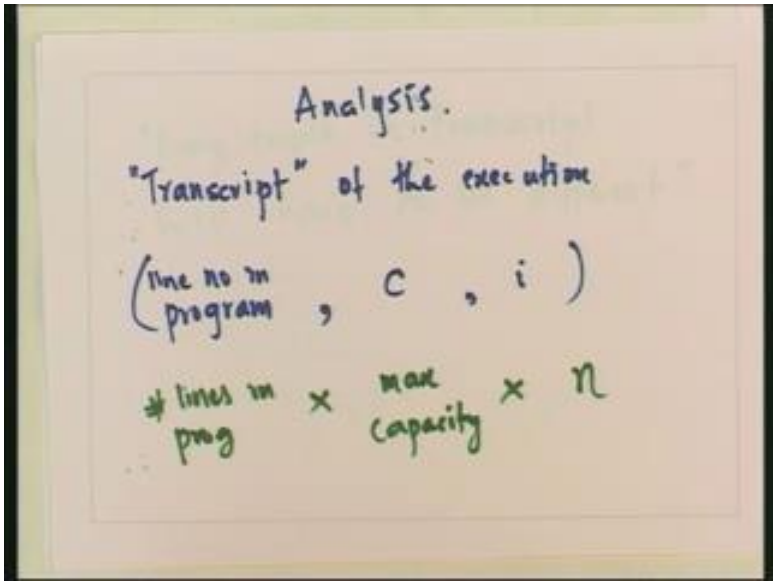
And this idea is, that every triple in this transcript must be different transcript will have to be different. So, what I am cleaning is that if this is the transcript that my computer wrote, then subsequently I am not ever going to see the entry 1, 10, 2 again. So, this is never going to happen, why is that the answer to that comes in the changes that we made. So, if some entry reappears, then that means the computer is executing that same statement, whatever the statement is with the same two previous parameters, same two values of C and i.

But then; that means, the Ks must have been called with exactly those two values C and i again twice at least. But, that was precisely the point of the changes that we made. So, we said that before we make a call to Ks, the computer actually checks. Did I already

execute this call, if I did execute the call, I am just going to pick up that value from my table. And therefore, we know that the computer never makes a called to Ks, twice with the same values.

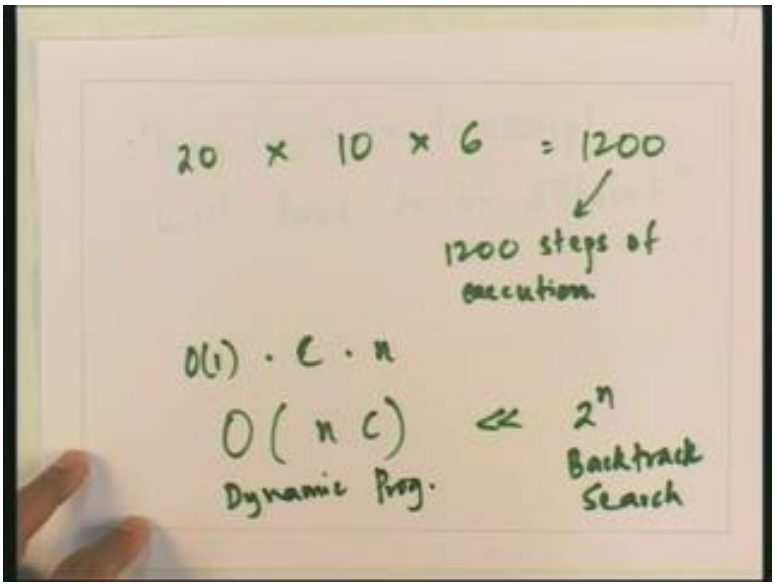
So, that from that it follows, the every triple in the transcript will have to be different. That gives us a good way to estimate the total length of the transcript. We just count, how many different triples can there be. So, let us go back to the place where, we wrote down whatever triples were. So, this is our definition of triples. So, since every triple has to be different, ((Refer Time: 45:12)).

(Refer Slide Time: 45:13)



The total length of the transcript will be, at most the number of different triples that are possible. So, the number of different triples that are possible is simply, the number of lines in the program, times the maximum value of the capacity times, the different values that I can take. The different values, that I can take n the number of objects. So, this is the number of entries that can there be in the transcript. So, let us complete our example.

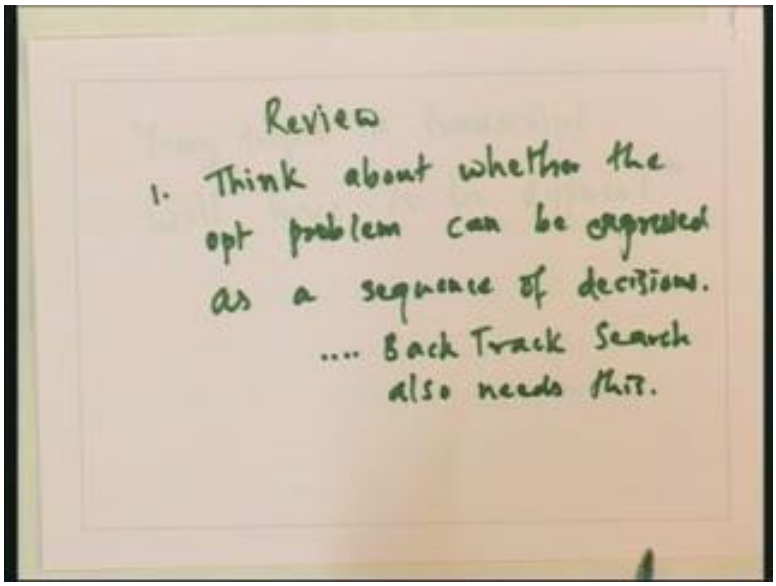
(Refer Slide Time: 46:16)



So, in our case the number of lines in the program, we just said was say something like 20 times the capacity we said was in 10 and n was 5, well actually we shall said n plus 1. Because, we allowed the program, the procedure to be called not with just the number of objects, but one beyond the number of objects as well, so in that case is 6. So, then we can estimate that this has to me 60 times 2 equals 1200. So, our program will require 1200 steps of execution.

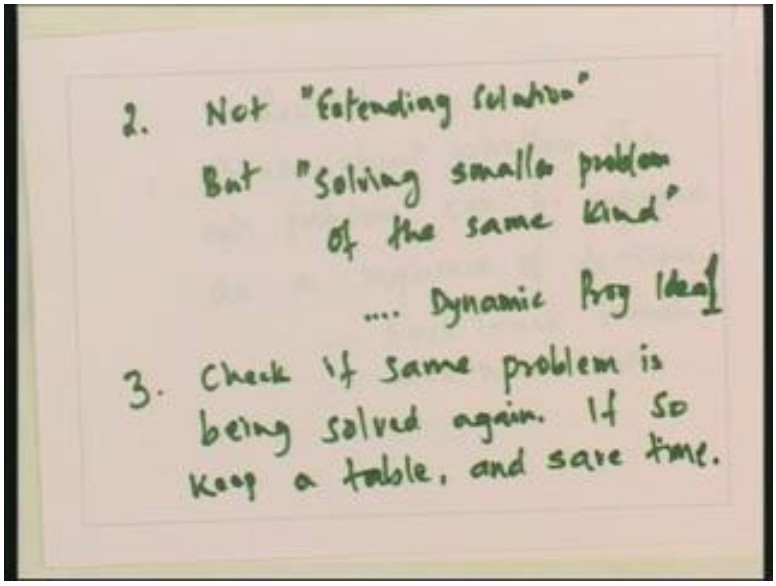
In general, the number of lines in the program, does not depend on the capacity that is given to you nor to the number of objects and therefore, I can write this as o of 1 times C times n or the total time taken is simply o of n times c, which is much, much less than 2 to the n, which was what we would get with backtrack search. And this is what, we get with dynamic programming. So, let me. So, let me now summarize the main ideas in all this, the main ideas in all this are 2 3.

(Refer Slide Time: 47:44)



The first idea is. So, let me just review this. This is the review; the main ideas are first think about whether the optimization problem can be expressed as a sequence of decisions. This is something that we need even in order to do backtrack search. But, beyond that in dynamic programming we do something more.

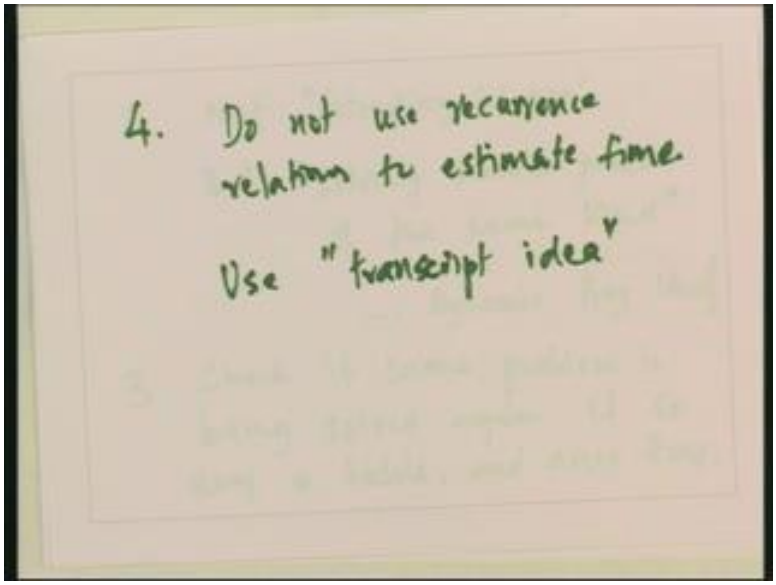
(Refer Slide Time: 48:46)



What we do more in dynamic programming is that rather than think of extending solution. So, do not extending solution, but solving smaller problem of the same kind, that is the important dynamic programming step, dynamic programming idea. Let me

see, this is the first dynamic programming idea. And in the second dynamic programming idea is, check if the same problem is being solved again if. So, keep a table and save time. So, you keep a table and we save time by not repeating that calculation. So, these are three ideas in dynamic programming, well. The first idea was really similar to was really common to backtrack search, but these are the two new important ideas.

(Refer Slide Time: 50:24)



And then there is also a fourth idea which is for the analysis. The fourth idea says that do not use the recurrences, recurrence relation to estimate time. But, instead use this transcript idea. So, these are the important ideas. So, let me say a little bit about what we are going to talk about in the next lecture. In the next lecture, we are going to use a slightly different formulation of what we have seen so far. And that will actually, end up eliminating the recursion.

And in some ways, it will simplify further our view of this whole procedure. And then we will also use dynamic programming, on some other problems. On these other problems, the expression and our program will become a little bit more complicated. But the basic idea as you will see will remain more or less the same.

Thank you.