Programming and Data Structure
Dr.P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 27
Algorithm Design-II

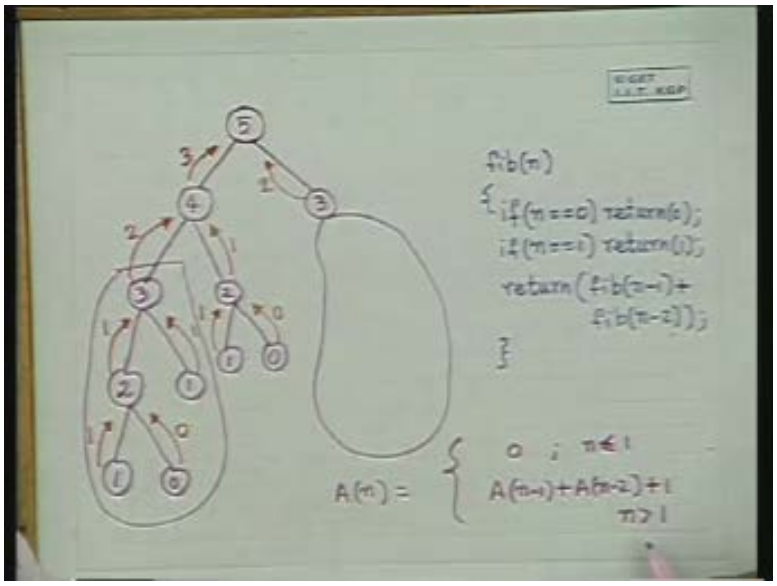
In the last class we have started discussing some algorithm design techniques and we discussed the concept of balancing where the recursive split was balanced to obtain the optimal structure and there we studied how to minimize the complexity of that balancing. And today we will discuss another issue and that issue is related to solving of identical sub problems and it leads to the concept of dynamic programming and we shall pick it up with traditional example and the old one which is the problem of solving Fibonacci numbers.

We have already seen that in order to solve Fibonacci numbers we have got the recursive equation is like this. The program is fib (n) if n is equal to 0, we return 0. If n is equal to 1 return 1 and otherwise we can return fib n minus 1 plus fib of n minus 2. This is the recursive definition of the problem. And in order to analyze whether this program is good or not, we will have to like in the split we tried to open up the recursive structure and see which one was most optimal. Here again we will open up the recursive structure. In order to analyze any such program we will have to open up the structure.

So let us see, let us solve it for fib 5. So to solve fib 5 you have to solve 4 and 3 and recursively you will first solve 4 and then 3, so you will come to solve 4, for that you will solve 3 and then obviously after 3 you will come to 2. You will come here, you will solve 2 then you will solve 1. Then for 2 you will solve 1 and 0, 1 is a base condition here and this will return a value 1. It will come here, it will call this, this will return a value 0. This will add these two and this will return here a value 1.

It will come and solve this, this will return a value 1. It will come here, this will return 2. Similarly it will come here and again you will stop solving this 1 to return a value 1 then come here 0, return 0. Then this will combine to return 1, this will combine to return 3. It will come here and repeat all these steps that this structure will be repeated here. The whole sequence will be repeated and this will return 2. So this is how we will be able to solve the Fibonacci number problem and also we have also seen to develop a recursive equation to find out how many additions you will do. The number of additions we saw, the number of additions we saw A(n) is equal to 0 if n is equal to, n is less than equal to 1. And it is the number of additions that you would do for n minus 1 plus the number of additions that you would do for n minus 2 plus 1 for n greater than 1. Isn’t it?

(Refer Slide Time 05:13)



This is what and we solved it in some class before and so this is possibly fib n plus 1 minus 1 or something like that. This is what it really does come to. And fib n, we know is an exponential number, so this algorithm is going to take exponential term. Now in order to analyze why this algorithm does not work polynomially is if you open this structure, the first thing that you will notice is that identical problems, sub problems are solved repeatedly. And this is what leads to this one, obviously is a base condition, so there is no question of repeating this. But 2 is going to be solved here once, here once and inside this also once because this structure is there. So 2 will be solved three times, 3 will be solved twice, 4 and 5 will obviously be solved once only but if you take a large number say 100, each of these lay around numbers will be solved again and again. So this is the core problem of such solving Fibonacci numbers.

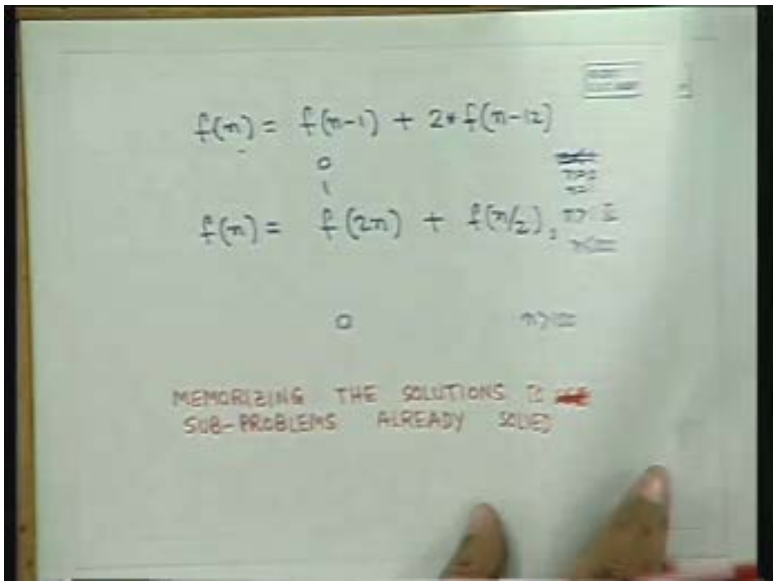
So how do we get about solving such problems, get about solving identical problems repeatedly? This problem has to be circumvented and in other situations where you come to a problem and you open up the recursive structure and see that a sub problem occurs multiple times and because you have written it out in a recursive fashion, this sub problem has to be solved repeatedly again and again. So how do we get about, how do we circumvent this problem and how do we resolve it? The method of doing so and solving a identical problem only once is the gross idea of dynamic programming in computer science language.

Now there are various ways of looking at it. First we shall see the most general approach and after we see the most general approach, we shall see how to adapt it to such Fibonacci numbers and how we can use this for other problems. For example instead of given a Fibonacci sequence, somebody may have given you f n is equal to f n minus 1 plus 2 into f of n minus 12 or you may be told f n is equal to f of 2 n plus f of n by 2. And you have got this conditions that this is for n greater than 1 and n less than 100 and n greater than 100 it is 0 and n less than 1 that is when n is equal to 0, it is 0 and it is 1 for n

equal to 1. Even in this you may be asked to find out and if you open this up you will see again identical sub problems will be repeating again and again. So how would you solve such recurrence equations or in general such recursive problems where you have got identical sub problems occurring many times?

So the first thing and the most obvious thing which would come to mind to solve such identical sub problems is memorizing the solutions to a sub problem solved. That is if you have solved a sub problem already and if you remember that solution you can just pick it up next time you come around. So, but in order to memorize all in order to remember, so in order to remember this you need to have some data area and this will be a global data area which you will have to remember.

(Refer Slide Time 10:02)

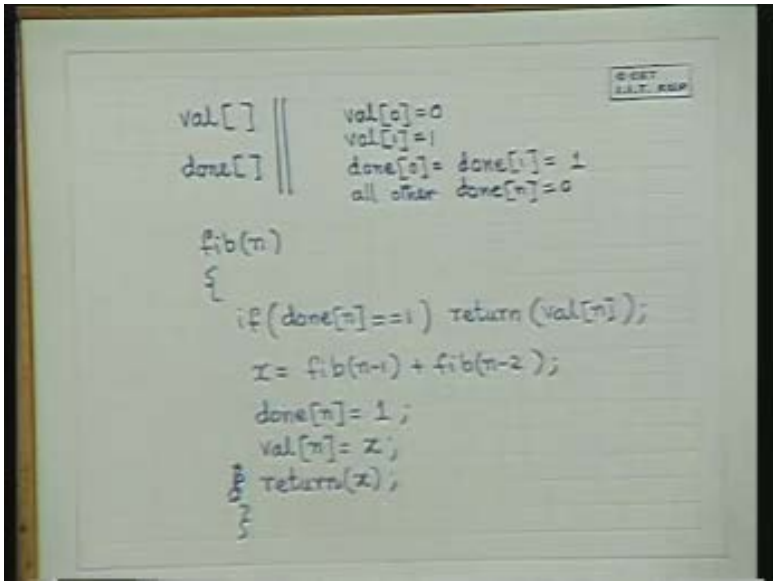


So given any such problem where you have to do and you know that you have to remember certain values, the simplest idea would be to maintain an array of those values. So let us see how would we do it here in the Fibonacci numbers problem. We would maintain two arrays, one is the array of f values and the other done which would be a Boolean array which would indicate whether this value is already obtained or not. So let us do two things, one is let us maintain a value array and let us maintain a done array. How it may be made dynamic etc etc, we will look at it later on. But if you maintain these two arrays, global arrays and initialize them like this.

We initialize them val 0 is equal to 0, val 1 is equal to 1 and done 0 is equal to done 1 is equal to true, so true is 1. So we initialize it like this and we write out our recursive structure again but with a slight modification. See then we need not bother about this conditions any more but even if you bother about base conditions it does not matter. Can anybody now tell me how we will solve this problem? If done[n] is equal to equal to 1 then return yes, return val n. Otherwise what would you do? X equal to recursively call it n minus 1 plus now what else would you do.

Yes, done [n] must be now made 1, okay done [n] must be made 1 and val [n] must be made z and return. Here also you must initialize all other done's to 0 that you must maintain all other done [n] is equal to 0.

(Refer Slide Time 13:00)

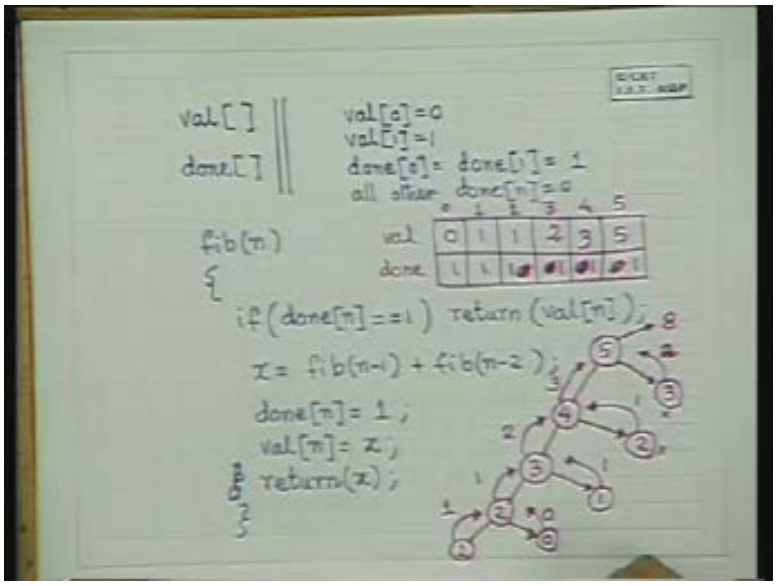


Now if you have done this then let us solve fib 5. To solve 5 we see first what val and done arrays are. Val 0 is 0, so this is val and done. It starts like this and all these will be 0. So to solve 5, you will see it is done. No, this is the array index 0 1 2 3 4 5. Done 5 is true, no, so you will call this. So you will call 4 and after you come back from 4 you will call 3. So you will come to 4, done is true. No, you will call 3 then 3 you will call 2, 2 you will call 1. When you come to 1, done is true so you will return val. So here you will return 1 then 2 will call this part 0, when it comes into 0 again done is 1, so it will return 0.

So now done 2 will be made 1 and val 2 will become 1 and it will return 1. When it comes to 3 it will call 2 again sorry, 1 again, 2 has done. Now 1 again, done 1 is 1, so it returns 1. Here now at 3 it will make done 3 1, it will make val 3 2 and it will return to 2. Here it will call 2 but when it comes here, it will get done 2 equal to 1 and once it gets done 2 equal to 1 it will just take val 2. Now, here this will make it 1 and this val will become 3 it will return here. It will call 3 done 3 is 1, so val 3 will be returned, val 3 is 2. Now done 5 will be made 1, the value of 5 will now become value of 4 plus 3, 5 and 8 will be returned.

So this way a problem which was solved before specially here and here is not solved again. So the total number of additions is now limited to this, this and this. So total number of additions becomes the number of values only but you have to make as many numbers of comparisons as well and you have to have additional data to store this.

(Refer Slide Time 16:00)



So for storing this information, we have reduced the addition from exponential to order n. The comparisons have also become order n but that's fine because the number of additions are now order n. So you have reduced the exponential time algorithm to a linear time algorithm by just remembering certain values. The same idea can be used in places like this where the range of n is known apriori. If the range of n is not known apriori then what will you do? You will have to maintain a data structure like a linked list for storing the values which you know. And what data structure will you maintain if the range is not known apriori? Suppose you have got arithmetic problems then you will not know the range of numbers then what data structure will you maintain. You can maintain a linked list of values or what do you have to find. You have to find out done n. So dynamic, you don't know which n's are now actually present, so you have to find n quickly and when a new n is done, you have to insert n.

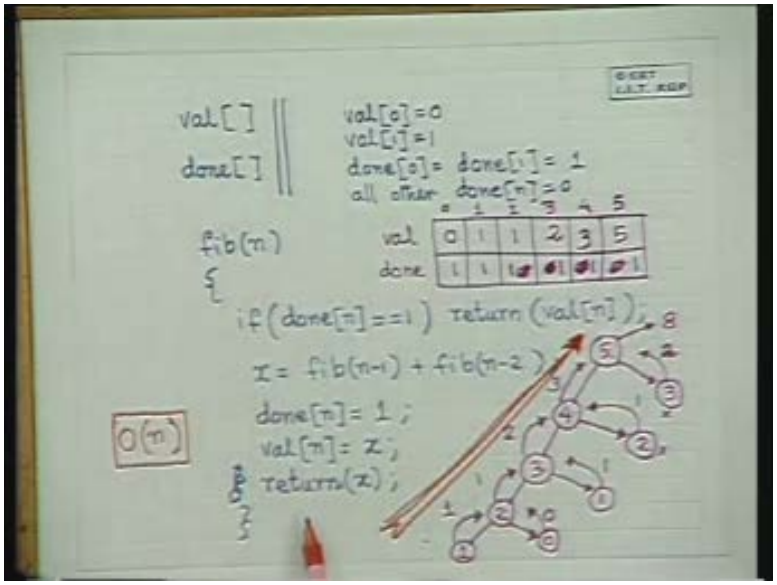
So your operations are insert and find on a set of numbers. So to do insert and find on set of numbers what will you, which data structure will you apply? A balanced binary search tree. So this is how you see that as and when the problem comes and as the data structures are known to you, you will be using data structures for algorithm design an algorithm design for data structuring and they go hand in hand to solve a large number of problems. So that is what I tried to say in the beginning that algorithm and data structuring are so interlinked that they go exactly hand in hand to solve a problem and you cannot separate one from another when you come to a particular problem.

So here you have seen that to do dynamic programming in an efficient way in many situations, you will need a data structure to store that memory that we were talking about, all right. And we have already seen that to solve many data structuring problems we need to solve a number of algorithm design problems. We shall come to it in subsequent classes. So this is the obvious way of solving a problem but if you want to solve a

problem in a more, there is another way of looking at it even deeper to see exactly how much you need to remember.

Do we need to remember everything? Yes or no. For that you will have to go in depth to see how the recursion proceeds and see how the values are nominated. Here though you have written them down in a top down fashion for 5 but see how the values are calculated. 5 calls 4 it does not compute the value of 4, it calls 3 2 1 and the values are computed in this sequence. So the values are, where the columns are made top down, the actual value computation goes bottom up. So if you have understood this definition which was the intuitive definition of the problem, if you have understood that what I have done has to be remembered and if you have understood in exact terms how this algorithm computes the value you might have as well throw away the whole thing and computed yourself.

(Refer Slide Time 20:03)

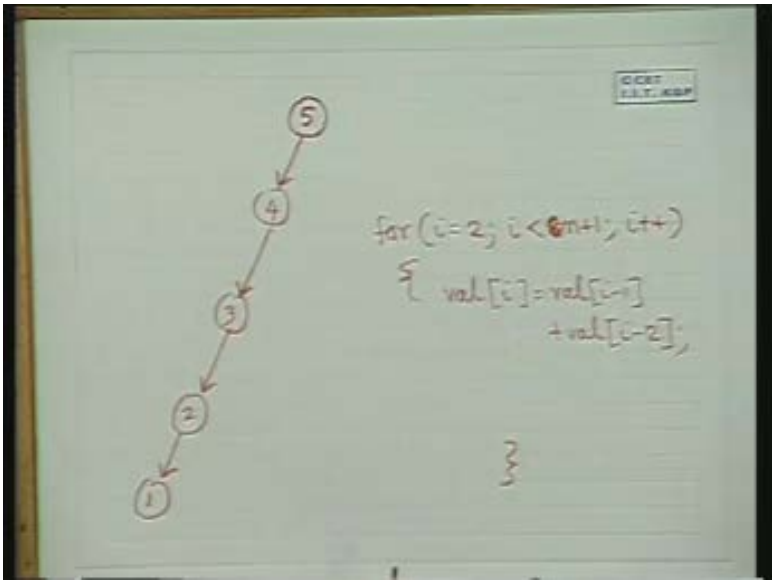


So the best way to compute this is to compute it bottom up yourself now because now you have understood how to do it. So we compute it backwards that is in order to compute 5, we know we have to compute 4, we have to compute 3, we have to compute 2 and we have to compute 1 and then we know 1 and 0. So we might as well compute two backwards up because that is the data dependency.

The data dependency is obvious here but it may not be obvious in problems like this. In problems like this the data dependency may not be so obvious, only when the data dependency is not obvious then use a done array and val array then your problem is solved but if you can go further and find out the data dependency then you can work it out according to your dependency in your own way. How would you do it? Solve it bottom up for i is equal to 2, i less than 6 or n plus 1, i plus plus simple, val i depends on i minus 1 and i minus 2 and if you are solving bottom up you know that when you are

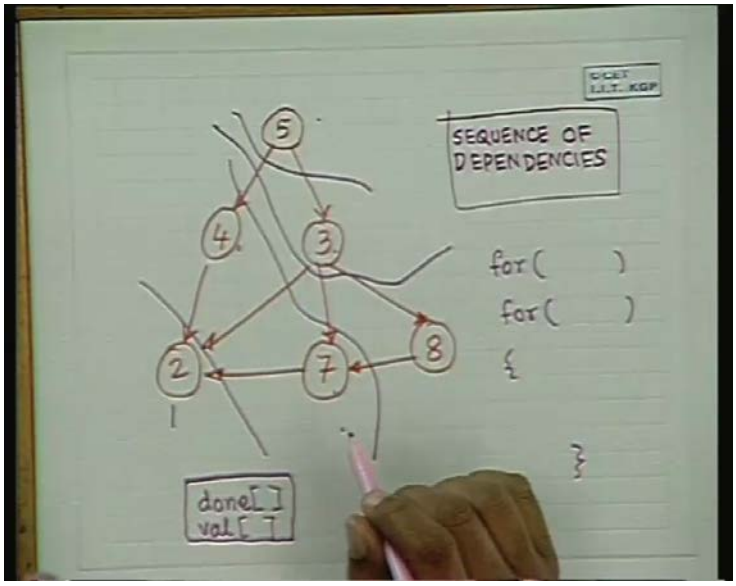
solving val i and val i minus 1 and i minus 2 are done and you need not check they are already done.

(Refer Slide Time 22:03)



This is a simple example where this comes obviously. So in other example where you know the structure you will see what is the approach to be followed. But let us see here it is not this and the dependency structure is something like this. Now suppose your dependency structure was in some other problem, suppose your dependency structure was like this then you would have to compute it in what sequence? This one first, so you would have to do a numbering this one first then second you can compute this or this or this. Can you compute this because this depends on this.

(Refer Slide Time: 24:06)



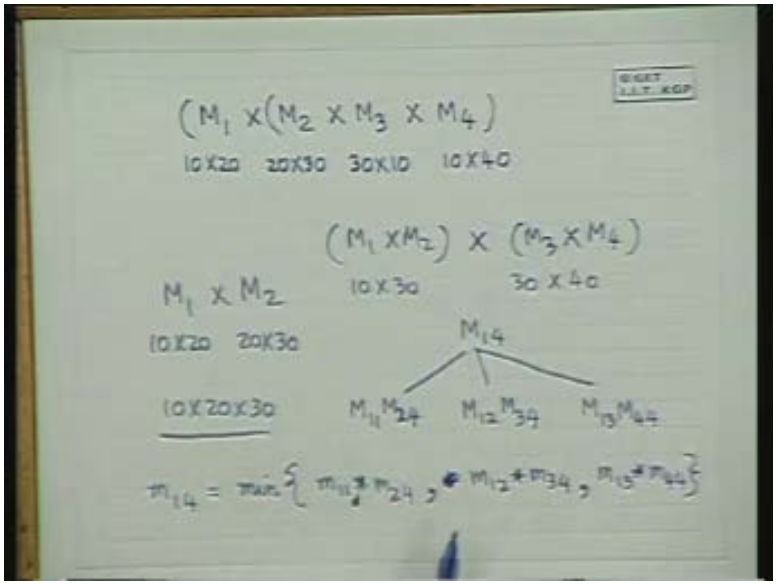
So you could have computed this second or this second. So this is the first which you compute, this is the second layer which you compute, this is the third layer which you compute, this is fourth and this is last. So you need to find out the sequence of dependencies. And once you know the sequence of dependencies, you can forget about maintaining the done array, you can forget about the recursion and you can solve it based on the dependencies itself. So you will see various versions of dynamic programming. If a person can find out dependencies, you will simply write it file loops, one or more for loops.

In other situations where the dependency is not known, you will use a done array or a variation of that done array to remember what problem has been solved earlier and use the value itself. So dependency usage and remembering what you have already done before so not to solve this problem repeatedly is the idea of dynamic programming. And this idea is used in several applications. I will just give you some examples of problems where these are used.

Consider the problem of multiplying some set of matrices M_1. Now in order to multiply this 10 by 20, this must be some 20 by something so 30. This must be some 30 by say 10 and this may be 10 by 40. Now to multiply two matrices, if you use the simple algorithm M_1, suppose I multiply M_1 by M_2. This is, how many multiplications do I require? Multiplications are more expensive than addition, you require 10 into 20 into 30 multiplications, all right. So to multiply two matrices like this I require so many multiplications. Now I have got lot of alternatives to do. Matrix multiplication is associative, I can multiply this, these all and then with this or I can multiply M_1 cross M_2 with M_3 cross M_4. If I multiply this with this, I will be multiplying a 10 by 30 matrix with a 30 by 40 matrix and here I will multiply this and this.

Now depending on the sequence in which I multiply, my total number of operations, total number of multiplications will come. If I have to find out the optimal sequence, what will I have to do?

(Refer Slide Time 27:22)



I will have to find out all possible such associative combinations, all right. To solve this problem what are my choices? To solve from M_{14} my choices are with M_1 M_{24} or M_{11} with M_{24} or M_{12} with M_{34} or M_{13} with M_{44}. And then again 2 4, I will now have various choices and based on these choices which ever is the minimum that is the one I should take. And if you write out see, therefore m_{14} is equal to minimum of m_{11} multiplied sorry m_{11} multiplied by m_{24} plus m_{12} multiplied by m_{34} sorry plus comma, comma m_{13} multiplied by m_{44} and again recursively m_{24}, m_{11} is 0 multiply a matrix by itself or if the difference is 2 you know what is the cost, all right.

So multiplying these two we will be multiplying, this is 0, 2 to 4 you can find the optimal cost and then this will be 10 into 20 into 40. So this way you can set up the equations and you will see that when you have a large one, M_1 say 10 then you will have multiple occurrences of M_{25} will occur in many places in the recursion. If you draw the full recursion tree, you will see lot of identical sub problems coming up and this is a problem which is a very clear candidate for dynamic program. So such problem, there are various other problems which are candidates for dynamic programming which comes up when the opening up of a recursion leads to several identical sub problems to be solved and all of them can be solved by various grades of approaches.

The approaches are firstly you have to remember the problem that you have solved before, secondly you have to see yourself can find out the dependencies. If you can find out the dependencies then you can forget about the done array and just work it out in a loop and if you cannot find out the dependency, you can simply use the done array and

the recursion, at least the number of repeated problems, the sub problems which are identical will not be solved repeatedly, you will be requiring some extra space that's all.

So dynamic programming to conclude is very essential component of algorithm design like balancing which tries to optimize the recursion split or the way in which we decompose the sub problem. Dynamic programming tries to ensure that identical sub problems are not solved multiple times. Therefore dynamic programming like balancing is another very very important aspect of algorithm design and it comes into play in many cases and in both in many situations; you will see the combination of balancing and dynamic programming coming into the approach.