

Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 07
Module - 01
Lecture - 44
Dynamic Programming

In the next few lectures, we will look at a very powerful technique for designing algorithms called dynamic programming.

(Refer Slide Time: 00:08)

Inductive definitions

- Factorial $n! = n(n-1)!$
 - $f(0) = 1$ - Base Case
 - $f(n) = n \times f(n-1)$ $n > 0$
- Insertion sort
 - $\text{isort}([]) = []$ ✓ Base
 - $\text{isort}([x_1, x_2, \dots, x_n]) = \text{insert}(x_1, \text{isort}([x_2, \dots, x_n]))$

The slide also features a diagram for Insertion sort showing a list $x_1, x_2, \dots, x_n$ with an arrow labeled 'insert' pointing to the position where x_1 is being inserted into the sorted sub-list $x_2, \dots, x_n$. The word 'isort' is written above the list.

So, the starting point of dynamic programming is to consider, what we would call inductive definitions. Now, there are many very simple functions which we come across which are inductive. So, we all know that mathematically n factorial is n times n minus 1 factorial. So, we can write an inductive definition of factorial as follows. The base case is when n is 0 and in this case the factorial is 1, so f of 0 is 1.

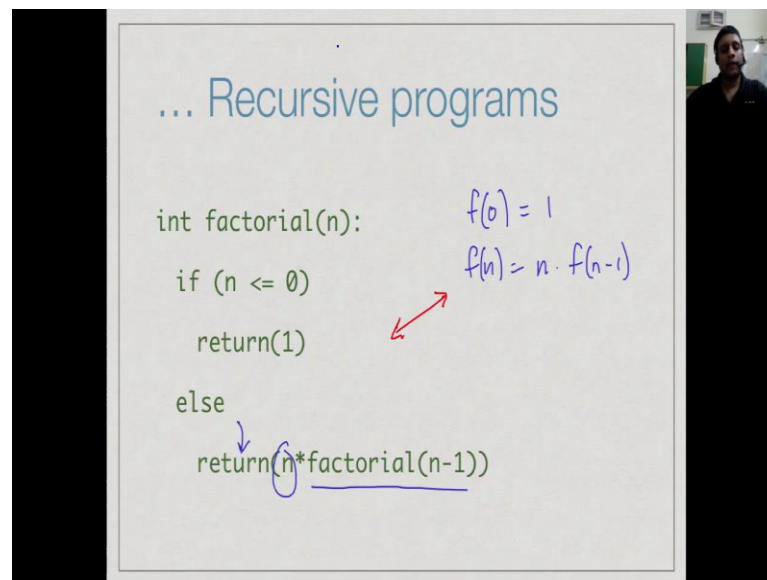
And in general, if we have n greater than 0, then f of n is given by n times f of n minus 1. In other words, we express the factorial of n in terms of the same function applied to a smaller input, now this kind of inductive definition is not restricted only to numeric problems, you can also do it for structural problems. So, for an example in a list or an array, you can consider a sub list or a sub array as a smaller problem.

So, here is a very simple way of describing insertion sort. So, if I want to sort n elements, then the way insertion sort does is that of course, there is nothing to sort the base case,

then we have done. Otherwise, we look at the rest of the list starting from the second element and we recursively sort it and then we insert the first element into the sorted list.

So, the insertion sort apply to X_1 to X_n , requires us to insert the value X_1 in the recursively sort to X_2 to X_n . So, again we are applying the same function that we are trying to define to a smaller input and in the base case, the smallest input namely the empty one, we have an answer which is readily available for us.

(Refer Slide Time: 01:57)



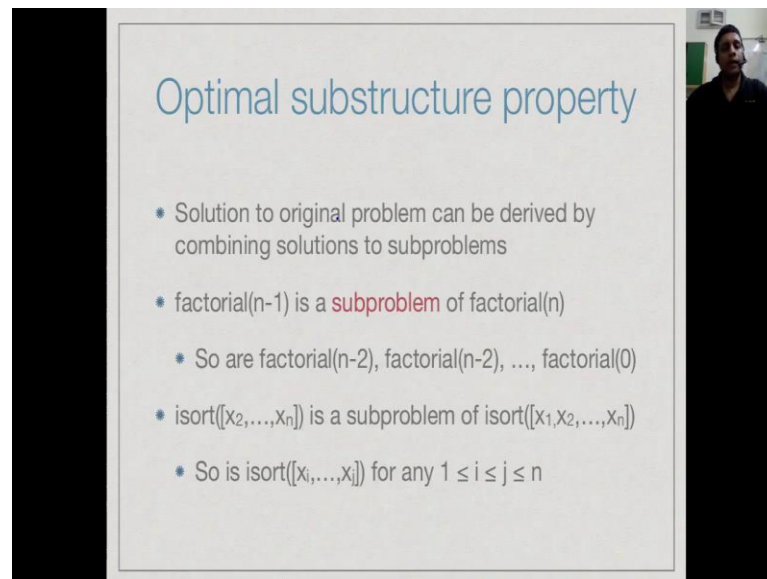
So, one of the attractions are having inductive definitions is that they yield in a very natural way recursive programs. So, we do not have to think much, we can almost take the inductive definition and directly translated it as a program. So, here is a translation for factorial which more or less reflex the structure that we had before, remember that the structure that we had before that was f of 0 is equal to 1 and f of n is equal to n times f of n minus 1.

So, we say if $n \dots$ Now, just to make it little more robust, so that people give, they give negative numbers, we get sensible answers, so just checking n equal to 0, you can just check for any value 0 or less, we will just return 1. If somebody ask if a factorial of minus 7, we just want to return 1. But the expected thing is that they will start with 0 and then if they give us a positive number which is bigger than 0, then we will go to the recursive keys.

So, we will compute factorial for n minus 1 multiply by n and then return this answer, well. So, there is a very direct one to one correspondence between this inductive

definition and the recursive program and that is what makes inductive definitions very attractive from the point of view of describing a function. Because, the inductive definition can be mathematically justified and then the program is obviously correct; obvious in codes, because it follows directly from the recursive of the inductive definition.

(Refer Slide Time: 03:17)



Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems
- $\text{factorial}(n-1)$ is a **subproblem** of $\text{factorial}(n)$
 - So are $\text{factorial}(n-2)$, $\text{factorial}(n-3)$, ..., $\text{factorial}(0)$
- $\text{isort}([x_2, \dots, x_n])$ is a subproblem of $\text{isort}([x_1, x_2, \dots, x_n])$
 - So is $\text{isort}([x_i, \dots, x_j])$ for any $1 \leq i \leq j \leq n$

So, what such inductive definitions exploit is what is sometimes called the optimal substructure property. So, what is complicated base means basically is what you expect from an inductive definition that is you solve the original problem by combining solution to sub problems. So, the solutions to the original problem are derived in terms of solutions in the sub problem and in particular to this sub problem is of same type, then it is computing the same type of answer.

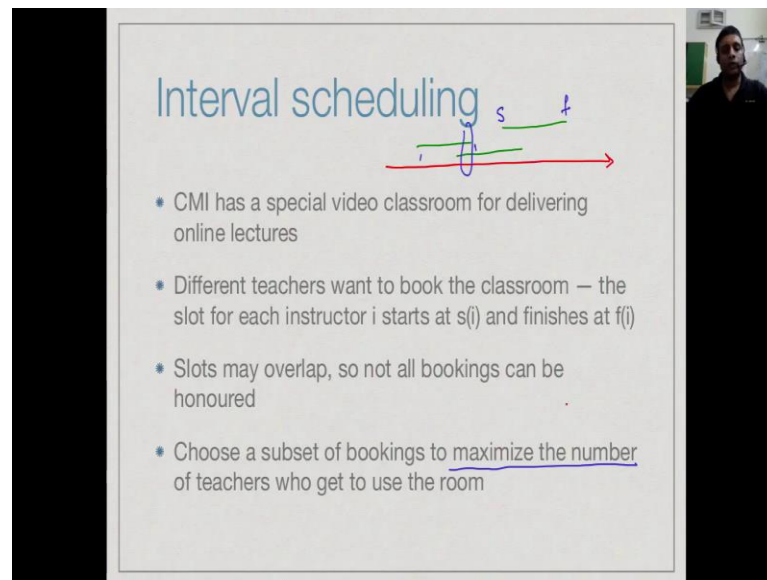
Now, in numerical question like factorial, it does not make sense to say something is optimal, but for example, when you doing insertion sort and certainly when you sort the sub list, the result of that is what you want for that sub list. So, this gives raise to a notion of a sub problem. So, factorial of n minus 1 is a factorial sub problem of factorial of n obviously, but it is just so happens at factorial of n , only required factorial of n minus 1.

Now, we could have problems which require more than one immediate is smaller sub problem, for instance factorial n minus 2 is also a sub problem, n minus 3 is also a sub problem and so on. So, any smaller input can be treated as the sub problem of the original thing. Likewise, when we actually do insertions sort, we give it the input X 1 to

X_n and we ask to sort X_2 to X_n . So, this is the sub problem which is directly part of the state.

But, in general one could think of any segment X_i to X_j to sort as a sub problem of sorting, this would happen for instances something like merge sort or quick sort, especially in merge sort. Then, you break up the array into halves and then into quarters and so on, so at any given point you are applying the same algorithm to some segment from A_i to A_j .

(Refer Slide Time: 05:05)



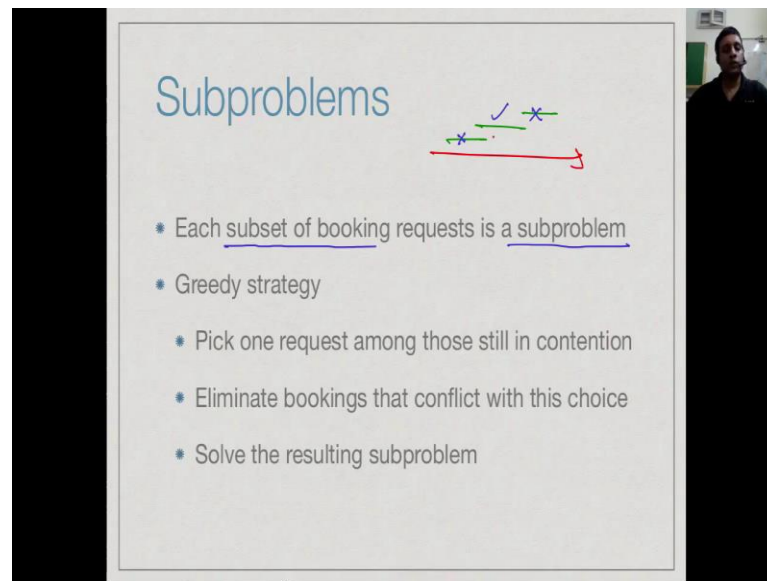
Interval scheduling

- CMI has a special video classroom for delivering online lectures
- Different teachers want to book the classroom — the slot for each instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured
- Choose a subset of bookings to maximize the number of teachers who get to use the room

So, now let us look at a problem that we are seen before in the context of greedy algorithms. So, we will look at the problem called interval scheduling. So, integrated scheduling said that we had a resource which is available over a period of time and now people will come and make bookings for these resources. So, somebody may want to book it during this segment, somebody else may want to book it in this segment, somebody else may wanted it during these segment and so on.

Now, during these overlapping things, you cannot give the resource to two people, so you have given a set of request each for the starting time and an ending time. So, we have a start and an end or a finish time and now what you want to do is, decide which of these requests you can allocate, so that the maximum number of bookings are actually granted. So, the goal is to maximize the number of bookings not the length of the bookings, but the number of bookings.

(Refer Slide Time: 06:03)



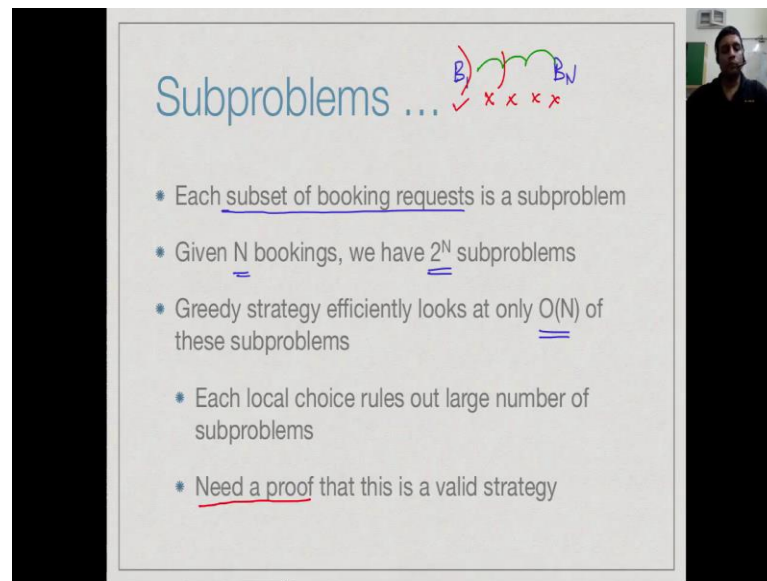
The slide is titled "Subproblems" in blue text. To the right of the title is a small diagram showing three horizontal lines representing intervals. The top line is green with a checkmark and an 'x' at its end. The middle line is green with an 'x' at its start. The bottom line is red with an arrow pointing to the right. Below the title is a list of four bullet points:

- Each subset of booking requests is a subproblem
- Greedy strategy
 - Pick one request among those still in contention
 - Eliminate bookings that conflict with this choice
 - Solve the resulting subproblem

So, in this particular case, what happens is that when you honour a booking, now if a booking happens to be overlapping with few other bookings, then if I decided to take this booking, then this goes away. So, these two bookings which overlapped with it, it can no longer be scheduled, because they are conflict with this in sometime interval. So, therefore now we have to solve or find a way of allocating the remaining bookings for some subset of the problem of the bookings.

So, each subset of the bookings is a sub problem in this case and the strategy that we saw was a greedy one, which said to pick the one which has the earliest finishing time. So, among all those bookings which has still available to us to allocate, we pick one in a greedy way by just looking at it among all those are remain the earlier finishing time. Now, this as we said, when we add, it will eliminate some bookings which are conflict that gives us a sub problem and you will solve these sub problem.

(Refer Slide Time: 07:06)



Subproblems ...

- Each subset of booking requests is a subproblem
- Given N bookings, we have 2^N subproblems
- Greedy strategy efficiently looks at only $O(N)$ of these subproblems
- Each local choice rules out large number of subproblems
- Need a proof that this is a valid strategy

So, how many sub problems are there? Now, we have N bookings and every possible subset of this is a sub problem, so we have an exponential number of sub problems. In general, we have any possible subset could be an answer. So, we have to look through all these exponential things in principle in order to find the best allocation, the one that gives the maximum number of bookings to be satisfied.

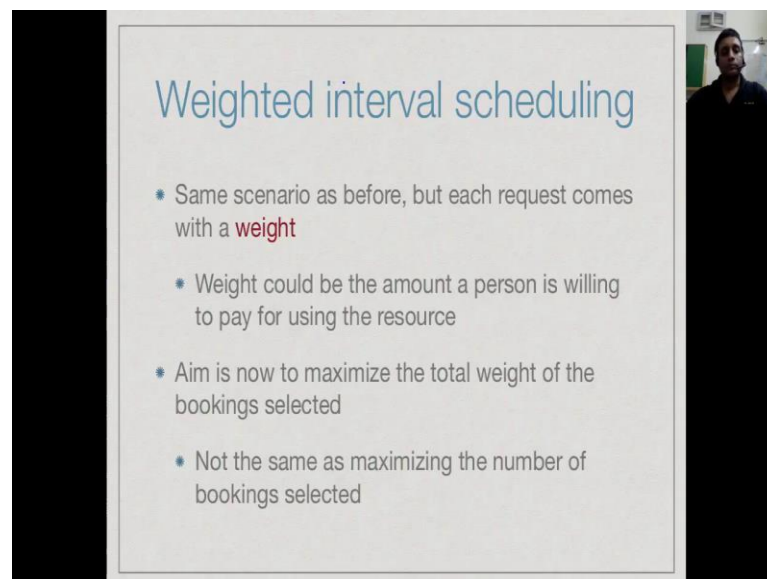
Now, what a greedy strategy does is effectively it cuts down this exponential space into a linear space, because what it does is it picks, so we have initially bookings 1 to N . Then, it will pick among these let us assume that are actually sorted by order of the earliest finishing time. So, you will take the first one, then you will rule out a few from there.

And now you will have some remaining and among those we pick the earliest one and you will rule out a few more and so on. So, at most you will allocate all N of them, but in each time once you rule out a few, you will rule out a few. So, certainly in a linear scan, you would look at only that many sub problems. So, you would look only order N sub problems and find, what you would claim is an optimal answer.

And since, you are doing such a drastic deduction from 2^N to the order N ; obviously, there is a question to ask whether you overlooked some sub problems accidentally by not examining them at all. So, you need a proof, so that is why in a greedy strategy, you need to prove that what you are doing actually makes the solution come out to be correct, because you are really not looking at a large number of, not considering a

large number of sub problems.

(Refer Slide Time: 08:39)



Weighted interval scheduling

- Same scenario as before, but each request comes with a **weight**
- Weight could be the amount a person is willing to pay for using the resource
- Aim is now to maximize the total weight of the bookings selected
- Not the same as maximizing the number of bookings selected

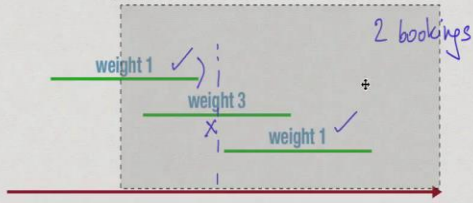
So, suppose we change the interval scheduling problem very slightly, we associate with each request a weight, a weight could be for example, the amount somebody is willing to pay, so may be people are trying to book, so we have an auditorium which we enter for performances and other activities. And people, who come to use it are willing to pay to use it. Of course, there is only one auditorium, so two people cannot use at the same time.

Now, our earlier goal was to maximize the number of bookings that we gave, but now, we have another criterion which is more immediate, which is how much each person is willing to pay. So, even if give to only one person, if that person is paying a lot more than everybody else and that would be optimum for us. So, now our aim is to maximize the total weight. So, we want to get as much revenue as possible from our allocation not the number of bookings, but the total weight.

(Refer Slide Time: 09:40)

Weighted interval scheduling

- Greedy strategy for unweighted case
- Select request with earliest finish time
- Not valid any more



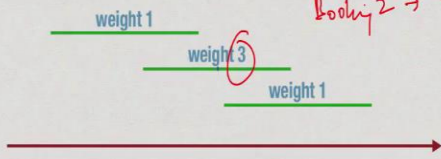
So, recall the greedy strategy in the earlier case, we wanted the earliest finish time. So, if we saw this particular selection of three requests, then the earliest finish time would be this one. So, we would first take this, that would rule out this and then because the third request starts after the first one completes, so you will take this and so we will get two keys. And these 2 out of 3 is the best we can do and that was find in the un weighted keys.

But, now unfortunately, what we have is that we have a weight, so we have this weight associated with this. So, we have to do something little more clever, because now if we choose the first one and third one, then the total weight is only 2.

(Refer Slide Time: 10:44)

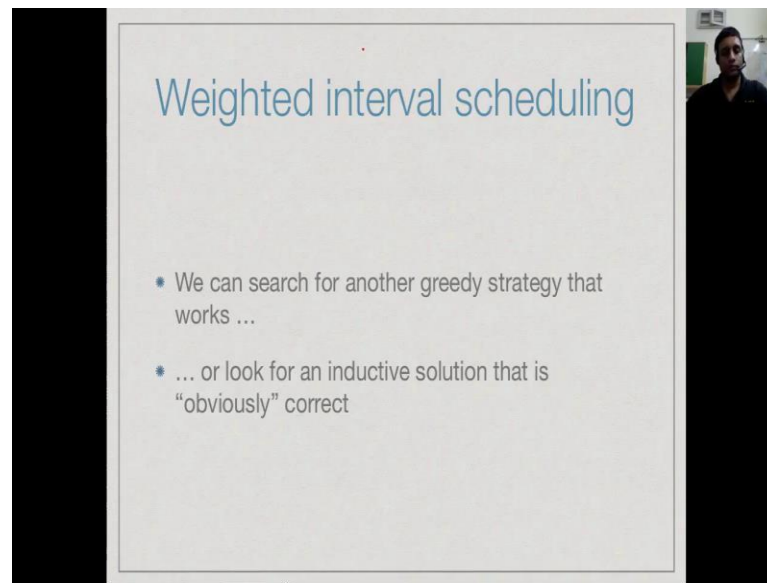
Weighted interval scheduling

- Greedy strategy for unweighted case
- Select request with earliest finish time
- Not valid any more



So, job, not job, but let us call it booking 1 plus 3 gives a weight of 2, whereas booking 2 alone gives a weight of 3, because it has a weight 3. So, ideally in this situation, we should recognize that the middle request has enough weight to overcome the penalty of it being the only one that will be scheduled. So, though we will one get out of 3 request schedule, we actually get a maximum benefit from the cost prospect. So, therefore that what just means in other words is the greedy strategy which we proved for the unweighted case is not valid any more unfortunately.

(Refer Slide Time: 11:27)



Weighted interval scheduling

- We can search for another greedy strategy that works ...
- ... or look for an inductive solution that is "obviously" correct

So, what shall we do? so one strategy is to see is there in other greedy strategy, we can search for another greedy strategy and try to argue that you would works and argue that it works as we saw is rather it takes a little bit of effort. Because, we have to use an exchange argument of some such thing to proof that is better than any solution that you could get by any others strategy.

So, the other approach which is what we are going to look at in more detail in the next few lectures is to try and look for an inductive solution which is obviously correct that which can be evaluated efficiently. So, the goal is to find to save, so what we are going to save is this effort in proving that by looking only at a few cases, we are actually producing an optimal answer. We would in some sense look at every case, but we look at every case in a clever way and that is what, we are going.

(Refer Slide Time: 12:22)

Weighted interval scheduling

Yes b_1 No b_1

- Let the bookings be ordered by starting time
- Begin with b_1
- Either b_1 is in the optimal solution or it is not
- If we include b_1 , eliminate conflicting requests from $b_2, \dots, b_N$ and solve the resulting subproblem
- If we exclude b_1 , solve the subproblem $b_2, \dots, b_N$
- Evaluate both options, choose the maximum

So, how do we do this for this problem is for this time, let us do something which is more direct, then what we did last time ones for looking at the earliest finishing time, just look at the earliest starting time. So, let us assume that our tasks are request or call order like this. So, we pick them up in this order. So, we are begin with the first booking which you called b_1 .

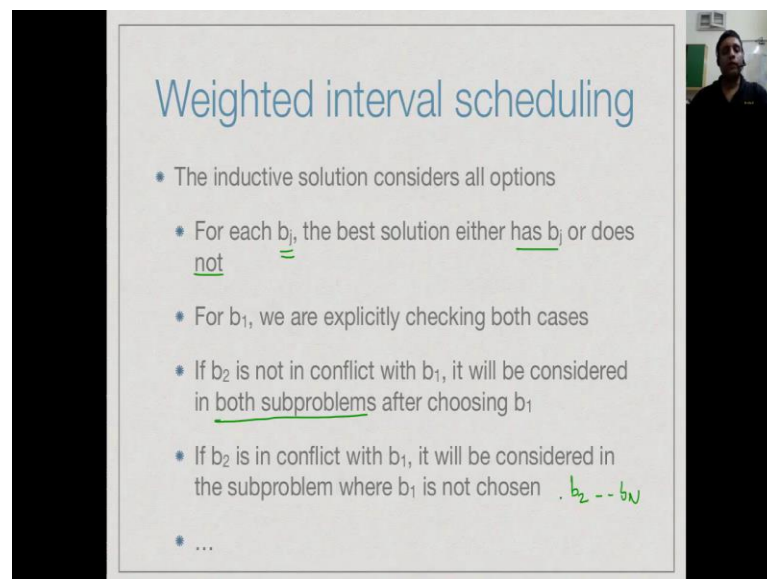
Now, observe that in the final answer, either b_1 is there, b_1 is not there. So, we will take two options. So, yes b_1 and no b_1 , now if we eliminate b_1 , then our sub problem just consists of b_2 onwards. So, we just have a sub problem which is b_2 to b_N . So, if we exclude b_1 , then we just use b_2 to b_N . So, exclude b_1 means just throw it out and pretend you only had $N - 1$ close to begin with.

On the other hand, if we include b_1 , then you have to be a little bit careful, because now if I include b_1 in this particular thing, so if we include b_1 , then we have to rule out something which is in conflict. So, we eliminate all the conflicting requests as we did in the greedy case and then we have another sub problem which is not necessarily b_2 to b_N , it will be a some sub set of b_2 to b_N .

But we are now taken both options, we are included b_1 and excluded b_1 , so it is more reasonable to expect that we have either a solution with b_1 or without b_1 , there are no third option, the solution either has b_1 and does not have b_1 , we are trying to evaluate both and then we are trying to choose the best one. So, this is an inductive decomposition of the problem with two sub cases with b_1 without b_1 , we are not making any

predictions about which is better, we evaluate both and take the better one.

(Refer Slide Time: 14:21)



Weighted interval scheduling

- The inductive solution considers all options
- For each b_j , the best solution either has b_j or does not
- For b_1 , we are explicitly checking both cases
- If b_2 is not in conflict with b_1 , it will be considered in both subproblems after choosing b_1
- If b_2 is in conflict with b_1 , it will be considered in the subproblem where b_1 is not chosen. $b_2 \dots b_N$
- ...

So, now let us argue that this kind of strategy actually considers all the options. So, just like b_1 for any b_j , the solution either has b_j or does not have b_j , this is very clear. So, there are 2^N possible solutions, I could either have b_1 or not b_1 , have b_2 or not have b_2 . So, I can try every possible subset that would be a group force argument, we want to avoid having to try every possible subset.

Now, for b_1 we have clearly checked both cases explicitly, what about b_2 , can we be sure that we are checking all cases where b_2 is? Let us look at b_2 . Now, if b_2 and b_1 are not in conflict that is b_1 and b_2 are in disjoint intervals, then whether or not b_1 is chosen is independent of whether or not b_2 is chosen. This means that whether we choose b_1 or b_2 the resulting sub problem would still allow to choose b_2 .

So, whether we choose b_1 or not, if b_1 or not b_1 at the beginning, it will be considered in both sub problem and when we solve that you will take the both choices. On the other hand, b_1 and b_2 are not comparable, that is b_1 rules out b_2 or b_2 because they overlapped. Then, when b_1 is chosen b_2 cannot be there, so b_1 can be there only if b_2 can be there only if b_1 is not there.

So, when b_1 is chosen we will not consider b_2 , but if b_1 is not chosen remember that we get the resulting sub problem b_2 to b_N . So, again b_2 will be chosen or given a choice, therefore b_2 we will consider all options in the presence or absence of b_1 . Likewise, we can argue that b_3 will be considered in the presence or absence of b_1 and

b_2 and what is happening as we are going along making a more on more commitments, we are ruling out lot of incompatible combinations which we would otherwise blindly considered, we get 2 to the N .

Now, will shall have to evaluate the efficiently, but the at least that is not that difficult to believe that we are actually trying out every possible option. We are not in advance deciding that some local criteria like in a greedy strategy is enough to rule out certain sub problems has been useless.

(Refer Slide Time: 16:33)

The challenge

- b_1 and b_2 in conflict, but both compatible with $b_3, b_4, \dots, b_N$
- Choose $b_1 \Rightarrow$ subproblem $b_3, b_4, \dots, b_N$
- Discard $b_1 \Rightarrow$ subproblem $b_2, b_3, \dots, b_N$
- Next stage, choose/discard b_2
- Discard $b_2 \Rightarrow$ again subproblem $b_3, b_4, \dots, b_N$

Diagram illustrating the conflict between b_1 and b_2 , and their compatibility with $b_3, b_4, \dots, b_N$. The diagram shows b_1 and b_2 overlapping, labeled "Conflict". $b_3, b_4, \dots, b_N$ are shown as separate bars that are compatible with both b_1 and b_2 .

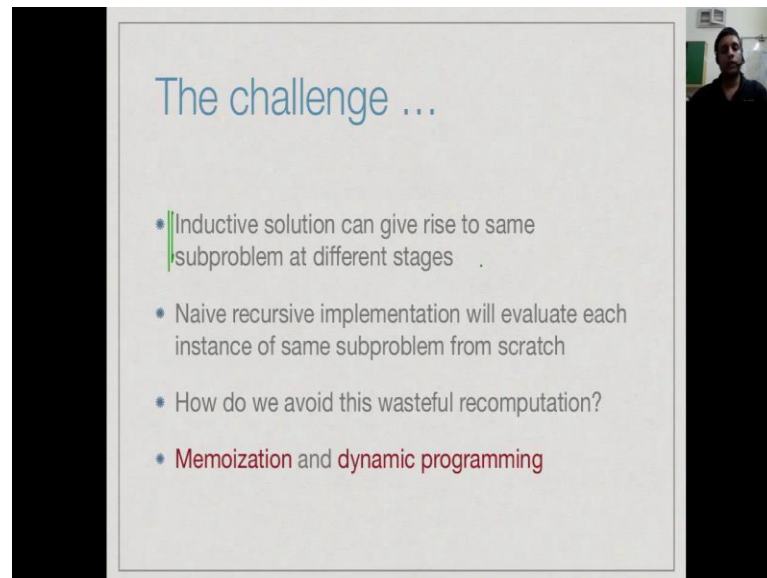
So, the computational challenge comes from the fact that the sub problems that we generate make appear again and again. So, let us look at a simple case, you supposing we are the picture that is shown below. So, we have b_1 and b_2 which are in conflict, but notice that both b_1 and b_2 are compatible with everything that comes after words. So, if we choose b_1 , then we rule out b_2 and so the sub problem, we get is b_3, b_4 and b_N .

On the other hand, if you rule out b_1 as we said before you will try out everything that remains namely b_2 to b_N . Now, what happens you need to try b_2 to b_N , so now, when you come to b_2 to b_N , you have to destroy b_2 or you have to keep b_2 . So, supposing you discard b_2 , then what happens when you discard b_2 from here, you precisely get the remaining part which is b_3 to b_N .

So, you again generate a b_3 to b_N problem which were already asked once in the context of disk of choosing b_1 . So, you now have that you have choose b_1 , say yes, no; if you choose b_1 I get this problem which is b_3 to b_N . Then, if I choose no, then I get a

chance to choose b_2 again yes, no and now if I do not choose b_2 . Then, I discard b_2 again I get b_3 to b_N , so I will be solving this problem once here and once here, unless I do something clever.

(Refer Slide Time: 18:09)



The challenge ...

- Inductive solution can give rise to same subproblem at different stages
- Naive recursive implementation will evaluate each instance of same subproblem from scratch
- How do we avoid this wasteful recomputation?
- Memoization and dynamic programming

So, the whole problem with this approach is that the inductive solution can give rise to the same problem at different stages. And if we just use recursion as we said before one of the great benefits are having on detective definition is that you can just write a recursive solution which just mirrors the inductive definition of the problem. But if you do it naively, every time you come to the function to be done inductively, you recursively call that same function, even if you have done it before and this can be very expensive efficiency.

So, the goal of dynamic programming is to avoid this wastefulness. So, there are two techniques that we will see, there is technique called memoization, which is a way to build in some cleverness into recursion. So, that you never call this same function twice recursively. And dynamic programming will then be a way to avoid doing this recursive calls all together. So, dynamic programming is a way to enumerate the sub problems directly and solve them, knowing that the sub problems have some dependencies which you can predict.

So, we will look at these two techniques, the next couple of lectures and look at several examples to get familiar with these notions of memoization and dynamic program, which are essentially ways of making inductive definitions at the corresponding recursive

implementations efficient to solve.