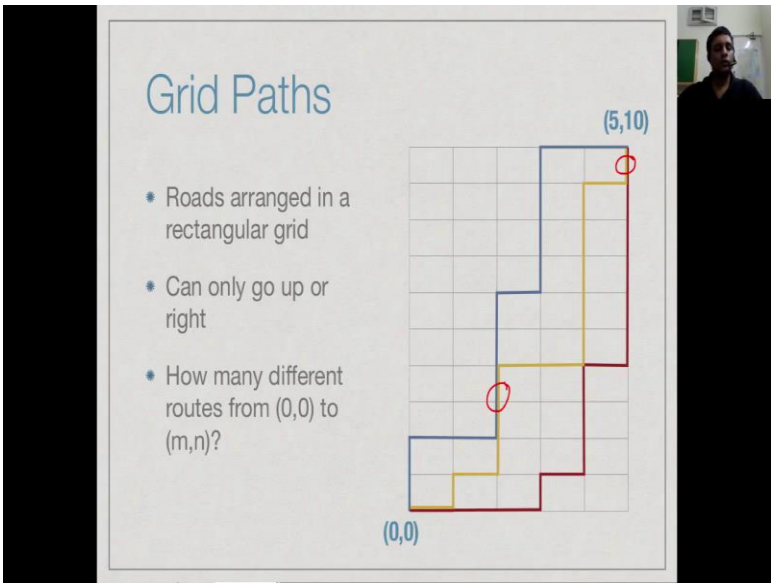
Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 07
Module - 03
Lecture - 46
Grid Paths

So, continuing our discussion about efficiently computing recursive functions, let us look at the problem of computing grid paths.

(Refer Slide Time: 00:09)

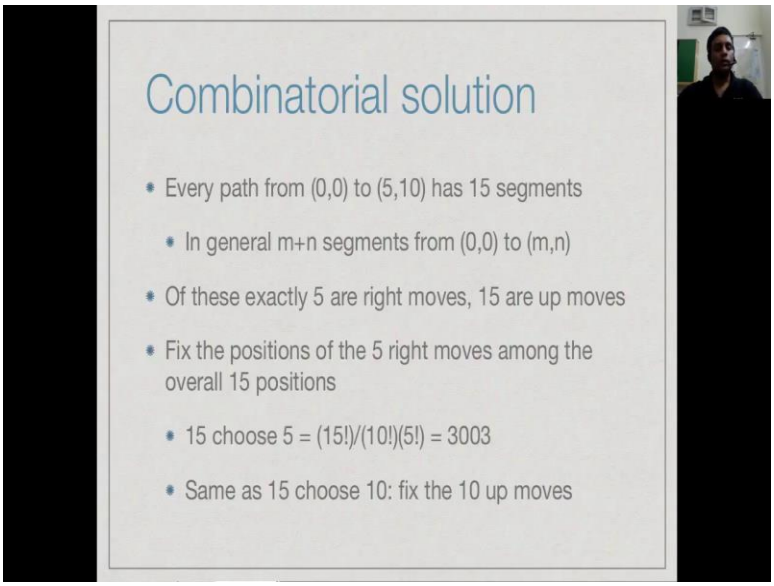


So, we have a grid, a rectangular grid, where we start walking at the bottom left corner, and the rule is that we can only go up or right. So, we want to start at the bottom and we want to reach the top right corner. So, we can number the coordinates. So, the bottom right corner we call (0,0). This particular grid has got 5 columns and 10 rows. So, the top right corner is (5, 10). And the question that we ask is how many different ways are going, are that go from (0, 0) to (5, 10).

So, what do we mean by different ways. Well, here for instance is one grid path. This blue line takes us up from (0, 0), then right, then up, then right and so on. So, this traces out one particular sequence of edges along this grid taking us from the bottom left corner to the top right. So, we could, of course choose a different one, which in this particular one the red one and the blue one are completely disjoint. They do not use any of the

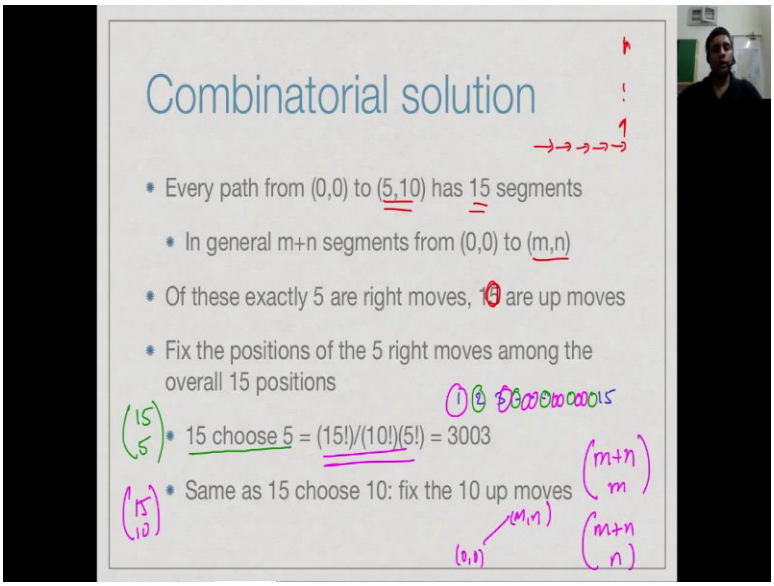
same edges, but in general, I could have paths which overlap with the other ones. So, this yellow one partly overlaps with the blue at the middle, over here, and then it overlaps with the red one over here, right. We want to know how many such different paths are there from (0, 0) to (m, n).

(Refer Slide Time: 01:28)



Now, it turns out this problem is actually a very classical problem in combinatorics counting. So, the way to analyze this is to see, that if I want to go from the bottom to the top, then I must, on the, in one-dimensional I must go from 0 to 5. In the other dimensions, I must go to 0 to 10. So, totally I must make 15 steps, right. There is no choice, I must walk 15 segments.

(Refer Slide Time: 01:51)



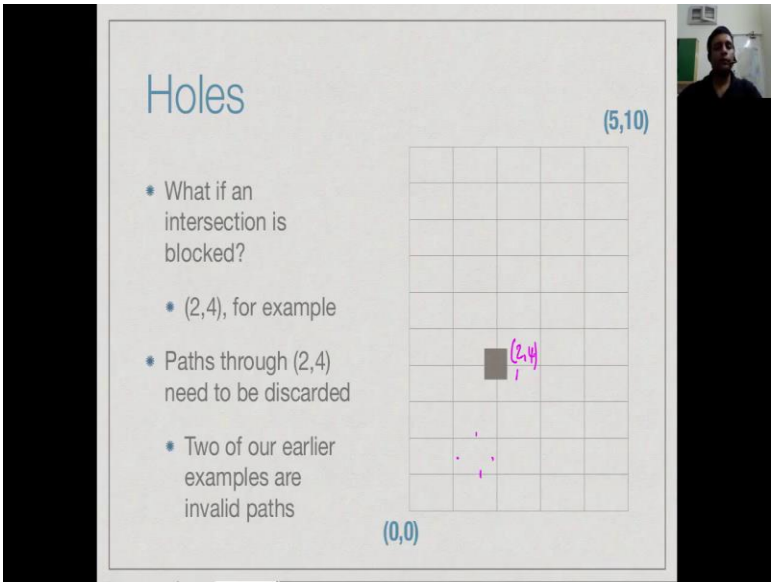
In general, if I am going to some (m, n), I must walk m plus n segments. Now, if I have these 15 segments and I am going to (5,10), that means, I have to go right, 5 times, right, and I have to go up 10 times. Now, in which sequence I do these rights and ups is what determines which path I take, right. But every path will have exactly 5 right moves and exactly 10 up moves, right. So, there will be 10 up moves and 5 right moves.

So, now if I take this and I think of this as an overall thing saying this is my 1st move, this is my 2nd move, this is my 3rd move and so on, up to the 15th move. Then if I tell you that you moved right, I moved right at these positions, at some five positions, then automatically I must have moved up at the remaining positions because I have to do exactly 5 and exactly 5 right and 10 up. So, therefore, all I need to do to determine exactly which path I am taking is to fix the position of the 5 rights moves among the total 15, right. So, among 15 positions I choose 5. This is usually written as 15 choose 5, right. So, this is a very standard, combinatorial thing, 15 choose 5 n choose k is n factorial divided by k factorial into n minus k factorial, right. So, 15 factorial divided by 10 factorial times 5 factorial, it happens to be 3003 ((Refer Time: 03:24)). So, for this particular grid there are 3003 ways.

Now, of course, instead of choosing the right positions I could also have told you the 10 positions where I moved up, that leaves 5 open positions where I ((Refer Time: 03:39)). So, this would give us 15 choose 10, and so it is not a coincident that 15 choose 10 and

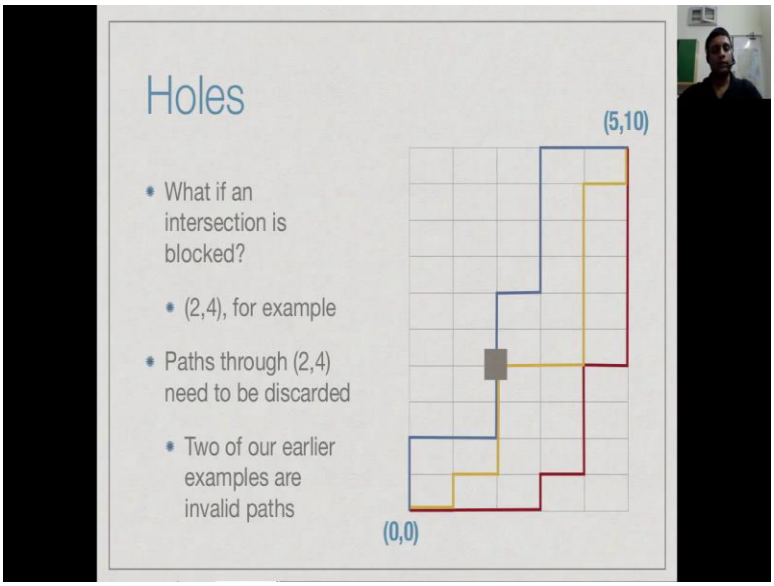
15 choose 5 are in fact the same expression. So, whether you choose to compute it as 15 choose 5 or 15 choose 10, it does not matter. So, in general it is going to be m plus n choose m or m plus n choose n where I am going from (0, 0) up to (m, n), right. I have to make m right moves and n up moves. So, I need to choose m out of m plus n total moves or n out of m plus n, both of them will give me the same expression, because that is why, n choose k, n choose k is n factorial k factorial n minus k. So, if you just exchange k and n minus k you get the same expression.

(Refer Slide Time: 04:31)



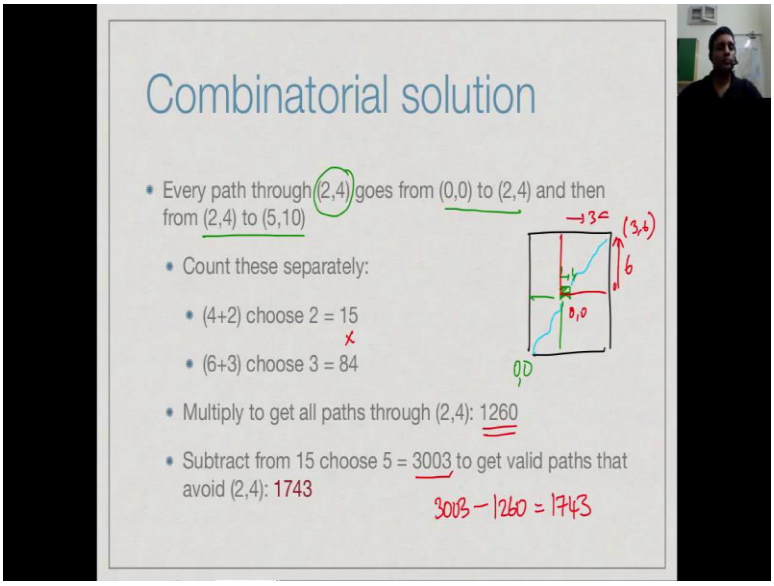
So, that is all very well, but what, for example, if this is not a perfect grid. Supposing we have some intersections, which are blocked. So, in this particular case, if you look at this intersection, which is (2, 4), right, it is 1, 2 and then 1, 2, 3, 4. So, we have put a black mark to indicate that for the moment, you cannot go through it, right. So, any part that goes through (2, 4), should not be counted among the valid path from (0, 0) to (5, 10).

(Refer Slide Time: 04:57)



So, the blue path that we had drawn before actually goes through this intersection. So, this path is no longer a valid path. The red path is ok, because it bypasses it, but the yellow path unfortunately also goes through this intersection. So, of the three paths, that we had seen so far, two actually do not go through. So, the question now is, out of those 3003 paths that we said were there, from (0, 0) to (5, 10), how many of them are still valid if you are not allow to go through this intersection.

(Refer Slide Time: 05:26)

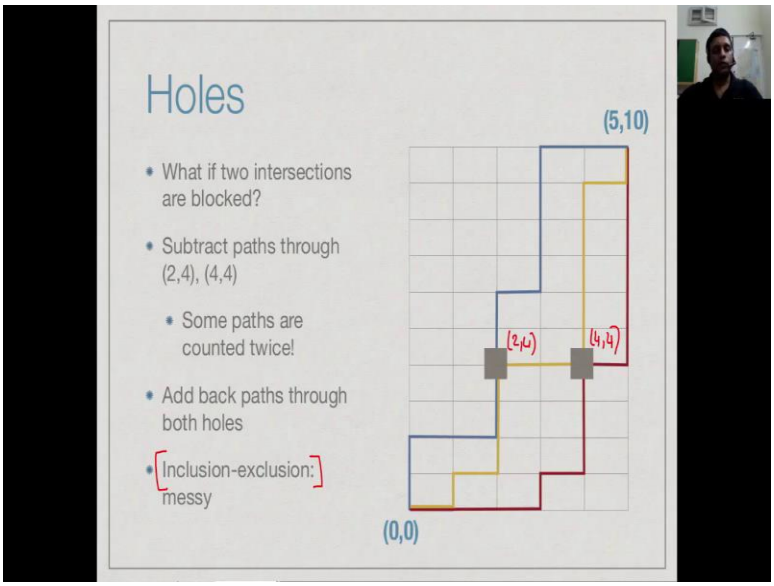


So, it turns over that actually this also has a combinatorial solution. So, what you can say is that in order to go from here and if I want to avoid a block intersection. So, let me see how many ways are there of going through it and just remove them, right. So, what I will do is, I will say, that every path it goes through the current block intersection is a path from (0, 0) to 92, 4) followed by a path from (2, 4) to (5, 10) because it goes through (2, 4). So, we can think of this as a smaller grid from (0, 0) to (2, 4), right.

So, if we solve, this is, this general m plus n m plus n choose n, right. So, I get 6 choose 2. So, I get there are 15 ways to go from (0, 0) to (2, 4). Likewise, if I consider this grid, then this is 3 and this is 6 because it was 10 and 5, I have done 2 and 4 respectively. So, after 2 there is still 3 left horizontally; after 4 there is still 6 left vertically. So, this is like going from a new (0, 0) to (3, 6), right. So, there I get 6 plus 3 choose 3 and this turns out to be 84.

Now, any part, which comes to the bottom 2 to 4 followed by any part, that goes from there to the top is a valid path passing through 2 to 4. So, I multiply these two numbers, I have to take 15 times 84 and again 1260. So, this is a total number of paths, which are going through (2,4), but I do not want paths going through (2,4). I am saying, that paths going 2 to 4 are not allowed. So, I must count all these parts as invalid. So, I take the original 3003 subtract. So, I take 3003 and I subtract 1260 and I get 1743, right. So, taking the combinatorial exercise to the next step. I can find out how many paths go from the origin to the given point on the top right provided one path one position is blocked.

(Refer Slide Time: 07:29)

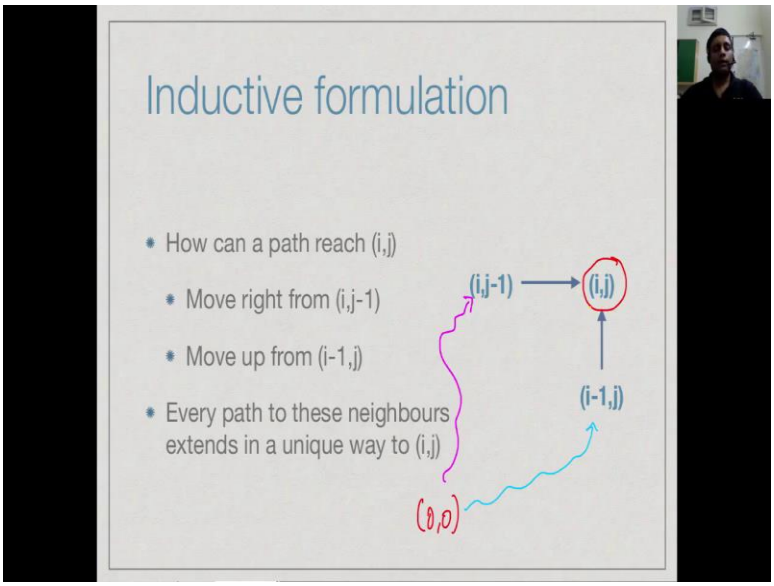


So, this imperfection could be more complex. I could have two positions blocked, right. In this case, the blue, yellow and red paths are all invalid because the red path also happens to get blocked. So, now I have blocked at (2,4) and also at (4,4), right. So, if I count all the paths going through two, (2,4), which I have already done, I can subtract this. Similarly, I can compute all the paths going through (4,4) and subtract those.

But now what happens is, this yellow path is subtracted twice because it is part of the paths going through (2,4) and part of the paths going through (4,4), right. So, I have accidentally removed it twice from my total, so I have to put it back. You have to now compute those paths, which go through both the intersections and add them back and this combinatorial is called inclusion and exclusion. You come across it also sometimes when you do Venn diagram, when you do sets.

If you want to find out, now you say, how many people are, how many sets have one element, how many sets are ((Refer Time 08:33)) intersection, how many sets you have, you know, you are, students taking three subjects, English, history and physics, and then so many taking English and history, so many taking history and physics and so on. So, when you do that kind of counting you will exactly do this inclusion exclusion, right. So, as we get more and more messy grids, this combinatorial question becomes more and more complicated to solve this way.

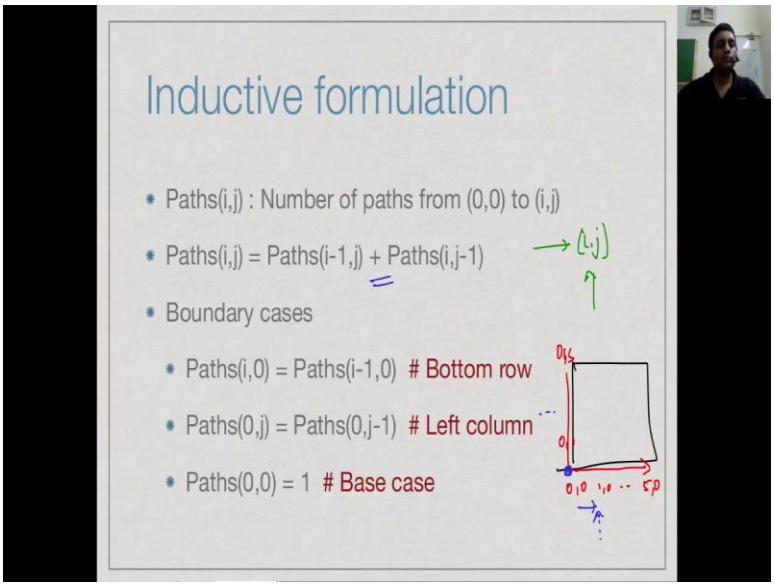
(Refer Slide Time: 09:00)



So, let us look for a better solution. So, as you might guess, since we are looking at these kind of inductive formulations and recursive programs, what we are really looking for is the inductive formulation of the grid path. So, let us ask ourselves the following question. How do I get to a point (i,j) on the grid?

So, I claim, that given our structure, which is, that we can go right or we can go up, there are only two ways I can come here. I can either come right from the neighbor on my left. So, from (i,j minus 1) I can make one step right and come to (i,j) or from (i minus 1, j), I can go up once there. So, if I have any paths, which starts at (0,0), right, any path, which somehow comes to (i,j -1), then by taking one, exactly one step, that path becomes one path to (i,j) right. So, every path from (0,0) to i minus, (i,j -1) can be extended in a unique way to (i,j). Likewise, any path, which comes from (0,0) to (i-1,j) can be extended in the unique way, right. So, if I count the task coming to the two neighbours, then each of those paths can be extended to reach the current node. So, therefore, I get the inductive formulation as follows.

(Refer Slide Time: 10:13)

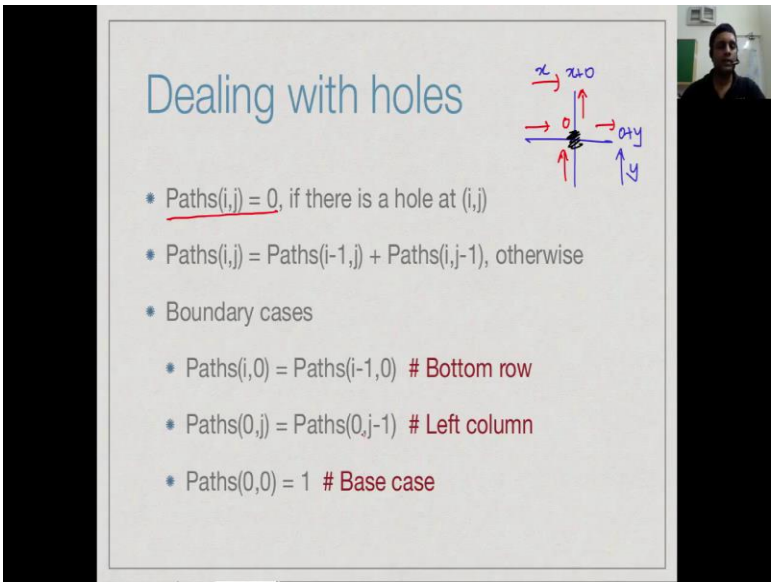


So, let us write paths (i,j) to denote the number of paths from (0,0) to the current point (i,j). So, what we have just seen from our inductive analysis on the problem is, that if you are at (i,j), then the paths come from left or from below. So, the paths to (i,j) are the sum of paths to (i minus1,j) and the paths to (i,j minus1). So, there are, of course, some boundary conditions.

So, if you look at our grid, in general, then if we look at the leftmost column, let us start the bottom row. So, we start the bottom row, right, then we know, that this is of the form (0,0), then (1,0) and so on to (5,0), right. So, anything of the form (i,0) derives its value only from the left because there is no corresponding row from the left. Similarly, from the leftmost column, from this (0,0), (0,1) and so on up to (0,15). And now, there is nothing from the left. So, I can only get it from j-1 from the row below.

And finally, you have to ask ourselves what happens at the initial conditions. So, if I am at (0,0) and I want to go to (0,0), how many ways are there? Well, this is a trivial path, there is only one way, by just staying there. It is important that it is not 0 because remember, that if it is 0, then everywhere you are just adding things and nothing will come, you will get no paths. So, it is important, that there is exactly one path from (0,0) to itself.

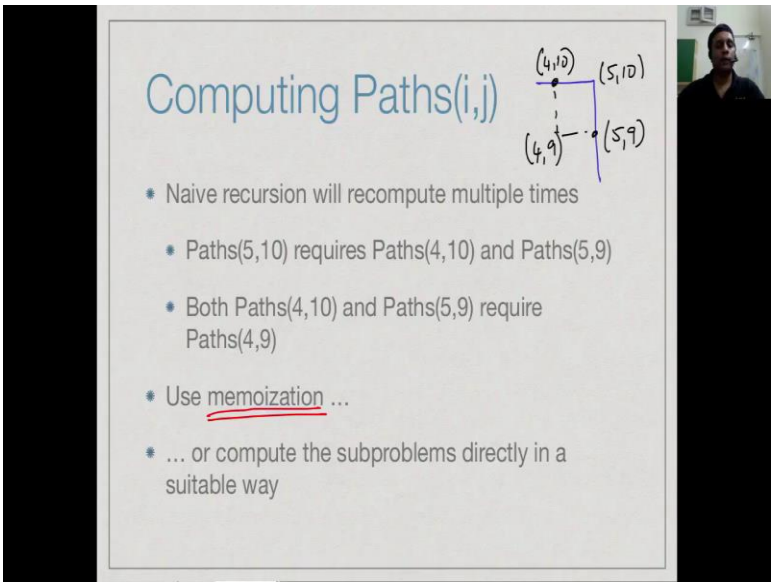
(Refer Slide Time: 11:42)



So, how do we deal with holes in this setup? That is easy, we just say, that whenever there is a hole at a given point we just declare parts of ((Refer Time: 11:52)) be 0. In other words, if there is a hole at some point, then this thing is going to contribute 0 regardless of what is above or below. So, if I come, if I have something coming from here and here I will just ignore it and say this is 0. And likewise, when I compute thing about there will be some value x coming from the left. So, this will just be x plus 0. And similarly, over here there will be something coming from below, say y, and this will be 0+y, right.

So, any hole will just have by declaration paths (i,j) equal to 0 because nothing can go through it and this will automatically propagate to its neighbors correctly. The remaining inductive ((Refer Time: 12:27)) exactly as before, right. So, if it is not a hole, it depends on its two neighbors. Then we have the bottom row left column and the origin as base cases.

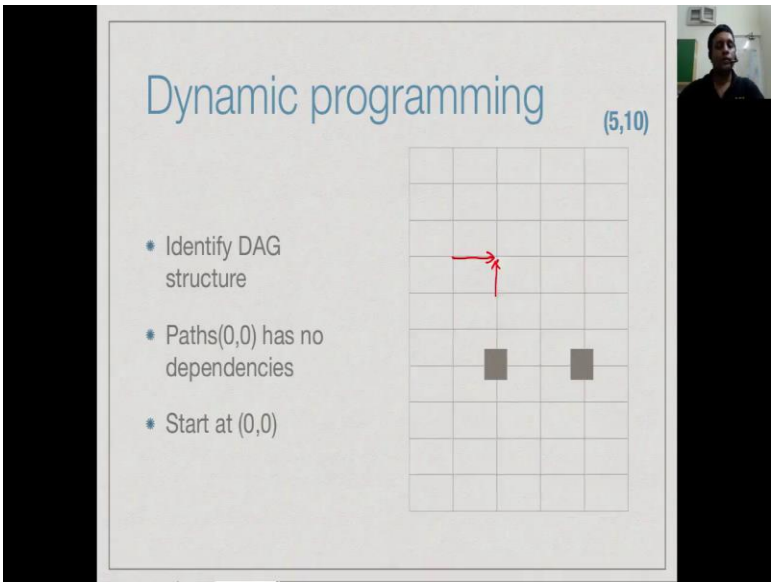
(Refer Slide Time: 12:38)



So, the difficulty with this calculation is the same as involved with the Fibonacci, which is, that if I have a particular calculation, say for example, supposing we start a recursive calculation at (5,10), then this will ask for (4,10) and (5,9). Now, these in turn will both ask for (4,9), so (4,9). if I just evaluate this paths function recursively, the way it is returning, the inductive definition, it will end up calling paths of (4,9) at least twice from this ((refer Time:13:12)) and this wasteful recomputation will occur throughout and we will get an exponentiation number of calls to paths.

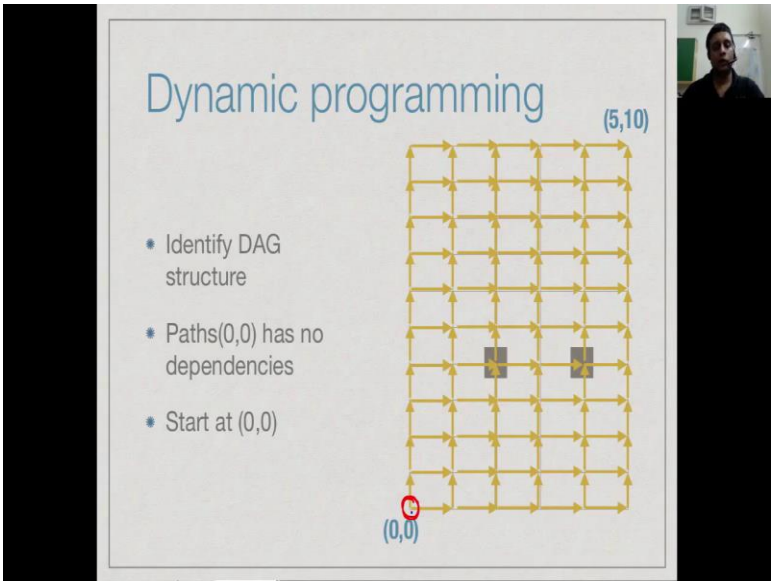
So, we have seen before, we have two technologies to deal with this. So, one is memoization where we just make sure that we have never paths (i,j) the same value of i and j more than once. The other way is to, is to anticipate the sub problems, figure out how they depend on each other, then solve them directly iteratively in a suitable order and this is what we call dynamic programming.

(Refer Slide Time: 13:42)



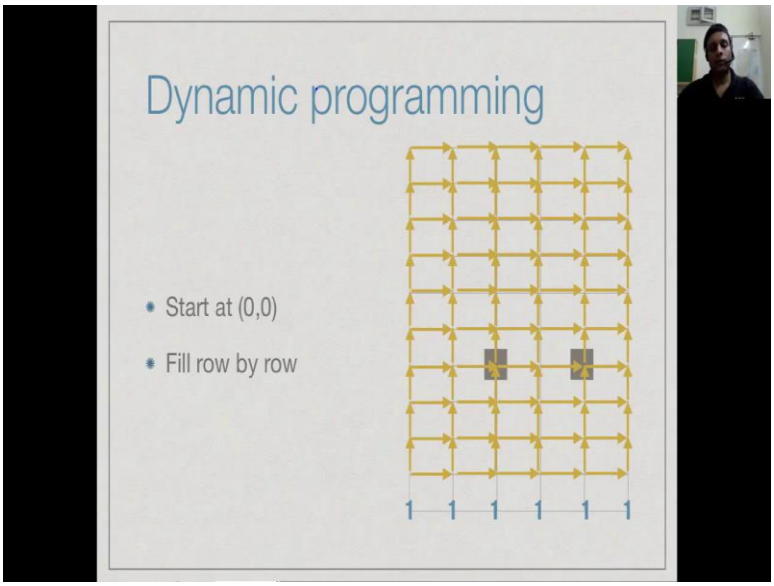
So, how would we use dynamic programming on the grid? So, this is our grid with the two holes at the (2,4) and (4,4). The first thing is to identify DAG structure of the dependencies, right. So, it is, we know, that every (i,j) depends on, it is left and bottom neighbor. So, this is how we do the DAG. If you remember, if these depends on the two values, left and below, we draw an edge from the left to this node and from below to this node.

(Refer Slide time: 14:10)



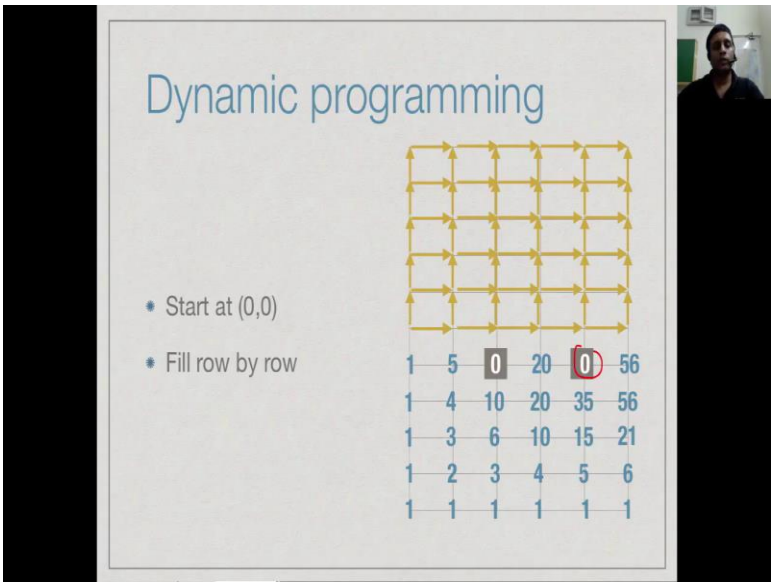
So, this is the DAG structure, right. So, ((Refer Time: 14:11)), the DAG structure. So, the DAG structure goes naturally from the bottom left to the top, right. So, this is the only 0 in degree node, this DAG. So, if you want to do a computation of this grid to grid paths directly using dynamic programming, the only place we can start is (0,0), right.

(Refer Slide Time: 14:35)



So, we start at (0,0) and we fill the value there, which is 1. And now, we observe, that we have two possibilities. We can go to the right or we can go up because these two dependencies are now removed. So, let us just go row by row. So, if we do this value, then automatically this dependency go. So, we will be able to do this value, when this will go, so we can do this value and so on.

(Refer Slide Time: 14:55)

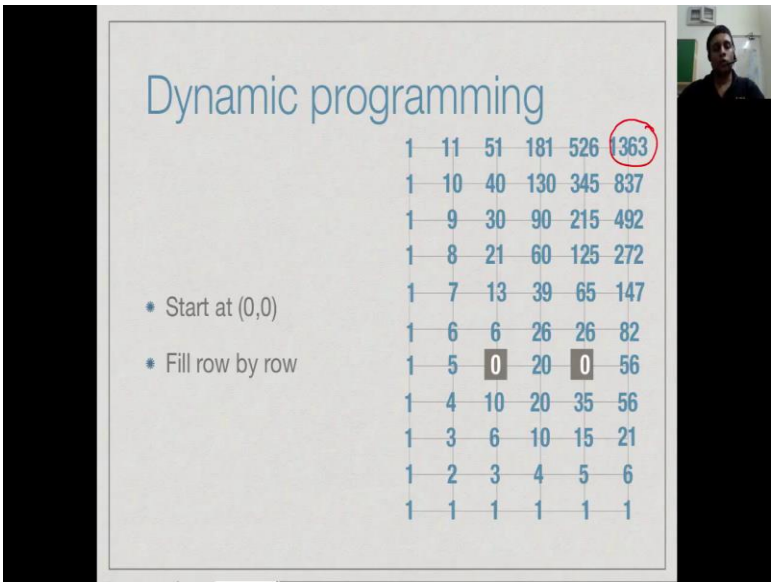


So, we can compute the entire bottom row and these are all inherited from the left using our base case, right. So, and it is very clear, I mean, intuitively there is only one way to get any of these places, which is just to go straight from left to right, no deviation is possible. So, there is only one path to every node at the bottom node.

Now, we can move up one row. We can see, that this node in the first row from the second row is now available. It is already available, but if we do that we can also do that ((Refer Time: 15:22)) and the node to its right and so on, right. So, we can fill the entire second row and at each point we are just adding up. So, 2s, 2 plus 1 is 2, 1 plus 3 is 4 and so on. Likewise, we can do the next hole. So, for example, 6 plus 4 is 10 and so on. Likewise, you can do the next row, 20 plus 15 is 35.

Now, we come to the holes, right. So, at the holes we said, that these will be 0 regardless of what comes from below. Even though there is 10 coming from below, this first hole must be 0, 35 from below, the hole must be 0. So, we just compute this row exactly as before except that the holes we input as 0 regardless of what is coming into it. We do not count it as 5 plus 10, we just put 0. We do not count this hole as 20 plus 35, we put 0.

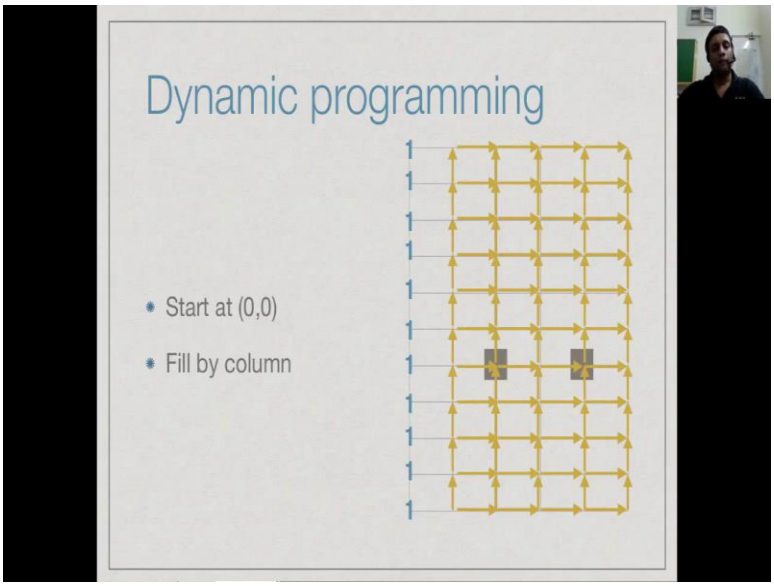
(Refer Slide Time: 16:04)



Now, when we go to the next row, likewise this 0 contributes only 0. So, above it, because no paths can come through this, this direction, the number of paths coming here is only 6 coming from the left. So, ((Refer Time: 16:12)) number of paths coming here is only 26 coming from the left. So, the number of paths, which are coming to this point could not come from there. So, this 20 is copied there, this 56 is copied. So, we can do this row by row.

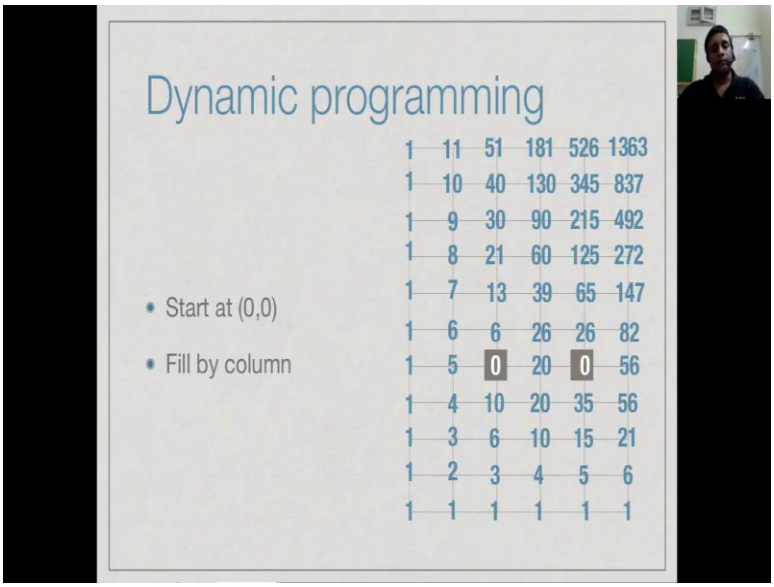
And if you can just enumerate every row like this by copying, adding up the values to the left and below and we will find, that there are actually 1363 paths with these two obstructions. And if we move these obstructions around, we can redo these calculations, it will be as efficient, we do not have to worry about this inclusion and exclusion. So, this is one way to do the dynamic programming, but remember any topological sort will work.

(Refer Slide Time: 16:48)



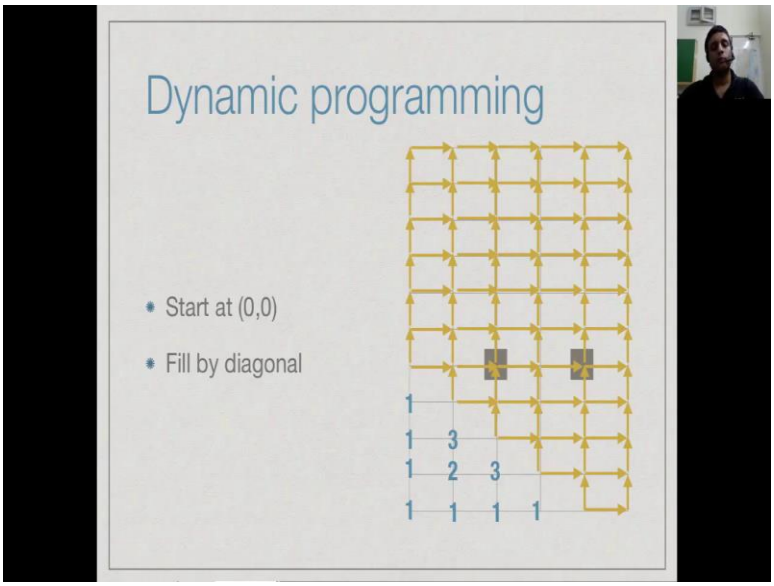
So, we could instead start with 1 and go column by column, right. So, we can go up and fill that and then we can fill up the entire first column. The first column will again be only once because there will be only way to do this.

(Refer Slide Time: 17:01)



Having filled the first column, now we can fill the second column and then you can fill the third column and the fourth column and so on. And obviously, this is only a different way of enumerating the values. The value is going to change. So, we should eventually end up with the same values at every point.

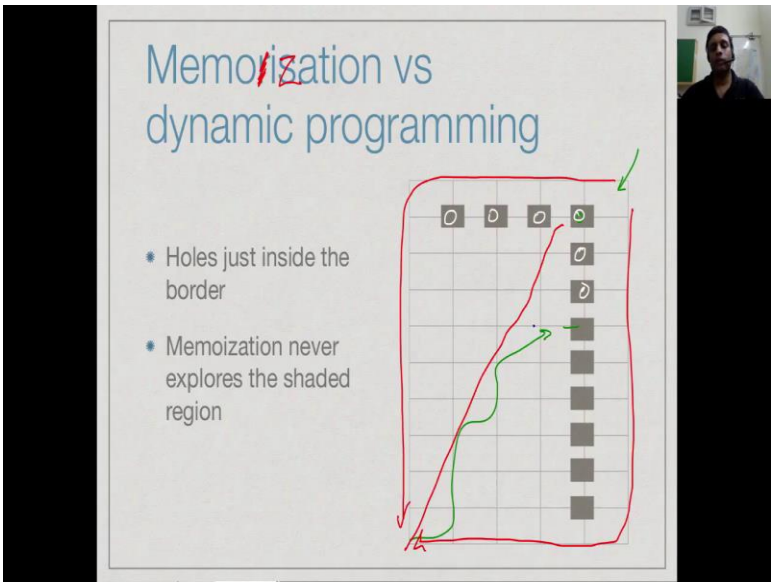
(Refer Slide Time: 17:17)



And finally, you can also do it by DAG. So, notice, that when I do this one, then these two points now are 0 and degree nodes. I can do both of them. Now, I have these three points, a 0 and degree mode. So, I can do all three of these. Then I have these four points, 0 and degree node. So, I can do all of these and so on. So, this is just to emphasize, that the choice of topological sort is entirely up to you. Very often, it will be more complicated to program this kind of diagonal things

So, usually what tries to do, it is a row and column, but any topological ordering of the, of the base values can be used in order to compute them. All you need to know is, when you come a node, the value you want to compute, all its dependencies must have already been computed, that is what this DAGS structure gives you, right. So, the sub problems form a DAG and you have to navigate your way through the DAG in a most effective way as far as programming it. It is usually by a row or a column in a table, but it could be the diagonal also.

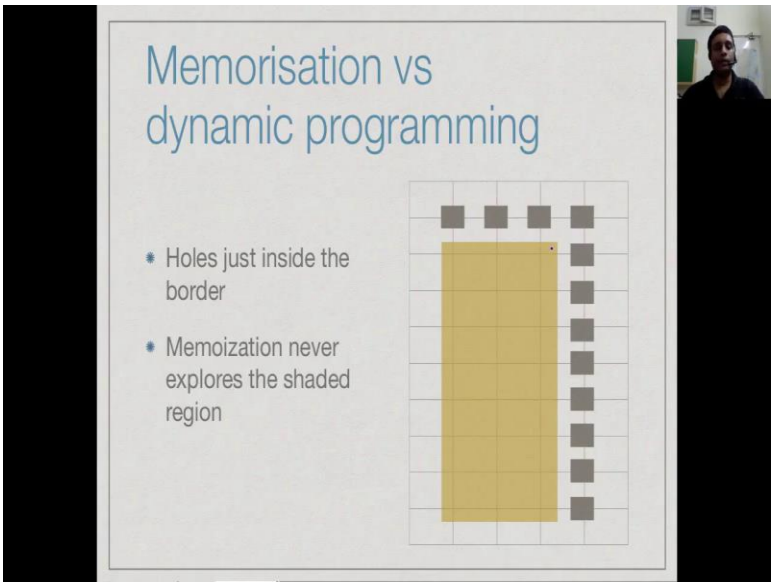
(Refer Slide Time: 18:18)



So, finally, before we conclude, let us just look at an instructive illustration of the difference between a memoization and dynamic programming. So, so if we have a bunch of holes which are along the border, then intuitively what it says is, that if I start from here and I go anyway like this, I am going to get stuck, ok. So, if I start my memoization from there, then everywhere it is just going to see a 0. So, all these values are going to be 0, and memoization is not going to look further around these.

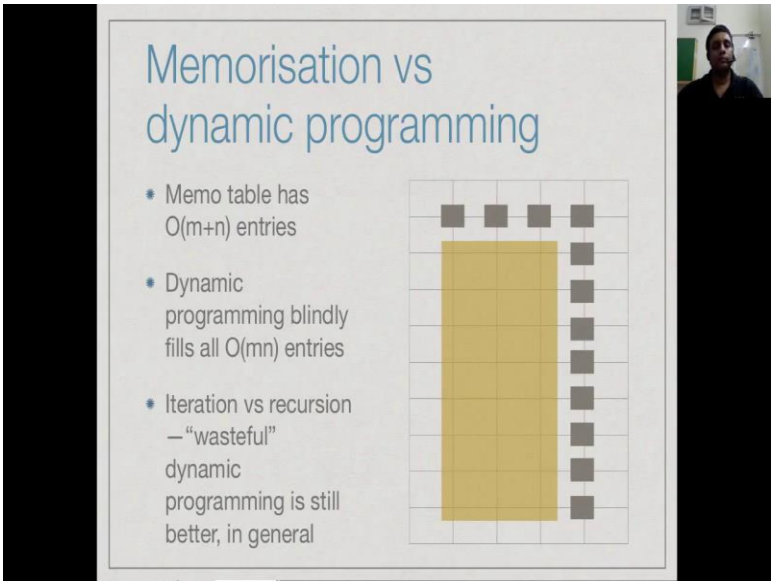
So, memorization, I claim, is only going to look along this outer boundary, only these grid points will actually be computed by memorization. Whereas, if I do dynamic programming, I will start form here and I would blindly fill up. Only when I reach the 0, I have realized, that the values are kind of computed inside the grid, do not worry, right.

(Refer Slide Time: 19:14)



So, there is this entire shaded region in this grid whose values are not needed because they cannot contribute to any path from, from the bottom to the top because every path going through them, you get blocked by one of these holes, right. Whereas, the dynamic programming is now going to blindly compute the values for these points even though they are useless memorization, will not because it only computes by need and it will never reach these points, right.

(Refer Slide Time: 19:36)



So, therefore, the memo table will have a linear number of entries, right. It will have, say, 2 m plus 2 n entries. It will only have the outer boundary of this grid ((Refer Time: 19:46)). So, it will have linear number of entries in terms of the two dimensions, whereas dynamic programming will have n times an entry will have. Every grid point will be computed in the table even though most of them are useless.

So, in a sense, in this example, dynamic programming is wastefully computing values, which we can by little bit of analysis realized will never be used because it is just blindly computing every sub problem on its way from the origin to the top. However, as we said before, dynamic programming is iterative. It is going to be just the simple, memoization is going to be recursive. It is going to be optimized to not make the same recursive call twice, but nevertheless it is recursive and recursion carries a cost in terms of execution in a programming language.

So, therefore though this looks like a wasteful dynamic programming strategy in case the holes are distributed in a bad way. Actually, it does not matter. It usually turns out, that a dynamic programming implementation will be usually more efficient than a memoization implementation. So, the memoized implementation is easy to do because we know, that we can go from an inductive definition directly to a recursive program.

And we saw last time, that there is a generic formula, a recipe to make any recursive program memorized. You just have to insert a couple of steps saying, look up the table and feed the table, right. So, therefore, getting a memoized implementation is very easy, getting a dynamic programing implementation requires some analysis of the sub problems and figuring out a good order in which to evaluate the sub problems. So, it is more work, but this extra work usually pays off, because then you can get an interactive computation of all the sub problems rather than a recursive one.