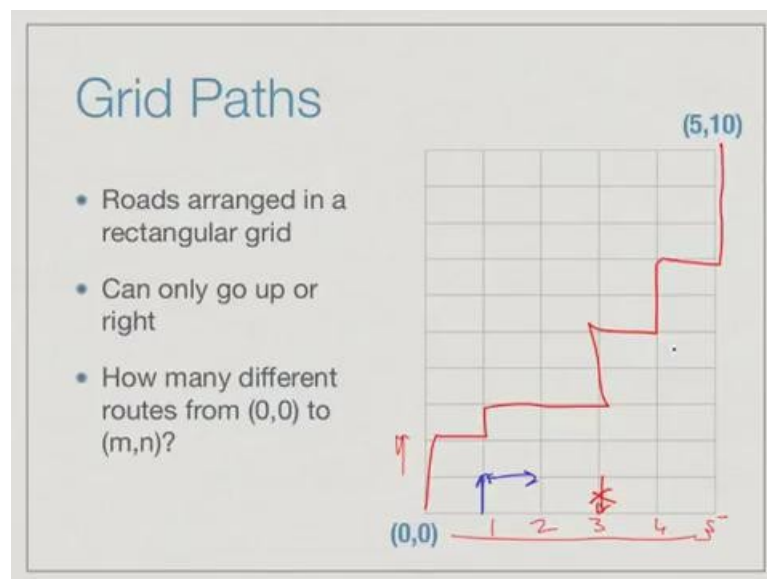


Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 08
Lecture - 02
Grid Paths

(Refer Slide Time: 00:02)

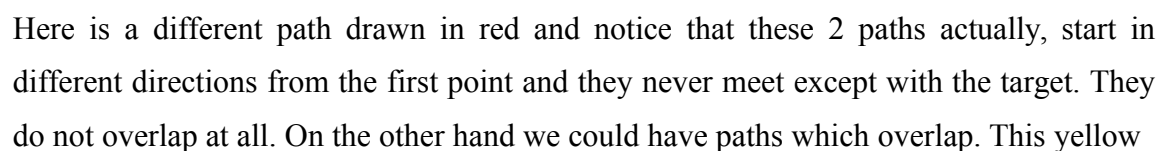
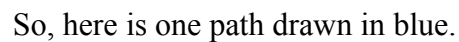


In the last lecture we looked at how to make iterative or inductive definitions more efficient than naïve recursion, and we saw memoization and dynamic programming as tools to do this.

Now, let us look at a typical problem and see how we can apply this technique. So, here is a problem of grid paths. So, we have a grid here, you can imagine there are roads which are arranged in a rectangular format. We can imagine that the intersections are numbered. So, we have $(0, 0)$ at the bottom left corner and in this case, we have $(5, 10)$ because going across from left to right we have 1, 2, 3, 4, 5 different intersections and 10 going up. So, we have at $(5, 10)$ the top right corner.

If these are roads the constraint that we have is that one can only travel up or right. So, you can go up a road or you can go right, but you cannot come down. This is not allowed. These are one way roads which goes up and right, and what we want to ask is how many ways there are to go from the bottom left corner to the top right corner. So,

(Refer Slide Time: 01:26)



path overlaps a part of its way with the blue path in this section and it also overlaps with the red path in 2 portions. There are many different ways in which we can choose to make this up and right moves and the question is, how many total such different paths are there?

(Refer Slide Time: 02:05)

Combinatorial solution

- Every path from (0,0) to (5,10) has 15 segments
 - In general $m+n$ segments from (0,0) to (m,n)
- Of these exactly 5 are right moves, 10 are up moves
- Fix the positions of the 5 right moves among the overall 15 positions
- $15 \text{ choose } 5 = \frac{(15!)}{(10!)(5!)} = 3003$

$$\frac{n!}{k!(n-k)!}$$
- Same as $15 \text{ choose } 10$: fix the 10 up moves

There is a very standard and elegant combinatorial solution. So, one way of thinking about this is just to determine, how many moves we have to make. We have to go from 0 to 5 in one direction and 0 to 10 in the other direction. So, we have to make a total number of 5 horizontal moves and 10 vertical moves, in other words every path no matter which direction we started and which move, which choice of moves we make must make 15 steps and of these 5 must be horizontal steps and 10 must be vertical steps, because they all take us from (0, 0) to (5, 10).

So, all we have to do since we know that these 5 steps are horizontal and 10 are vertical is to just demarcate which ones are horizontal and which are vertical. Now once we know which ones are horizontal we know what sequence they come in because the first horizontal step takes us from column 0 to column 1, second 1 takes us from one to 2. So, we cannot do it in any order other than that.

So, we have in other words we have 15 slots, where we can make moves and then we just say first we make an up move, then we make a right move then we make an up move then make another up move and so on. So, every path can be drawn out like this as 10 up

moves and 5 right moves and if we fix the 5 right moves then automatically all the remaining slots must be 10 up moves or conversely.

It is either 15 choose 5, it is the way of choosing 5 positions to make the right move out of the 15, and it turns out that the definition of 15 choose 5 is clearly the same as 15 choose 10 because we could also fix the 10 up moves and the definition is basically... if you know the definitions... then n choose k is n factorial by k factorial into n minus k factorial.

This k and n minus k basically says that 15 minus 5 is 10. So, we get a symmetric function in terms of k and n minus k . In this case we can apply this formula if you would like to call it that and directly get that the answer is 3003. There does not appear to be much to compute other than writing out large factorials and then seeing what the number comes.

(Refer Slide Time: 04:11)

The slide is titled "Holes" in blue. It contains a list of bullet points on the left and a grid diagram on the right. The grid is 10 columns wide and 10 rows high. The bottom-left corner is labeled (0,0) in blue. The top-right corner is labeled (5,10) in blue. A black square is placed at the intersection of the 2nd column and the 4th row, labeled (2,4) in green. To the left of this square, the numbers 1, 2, 3, and 4 are written vertically in green, corresponding to the rows. The bullet points on the left are:

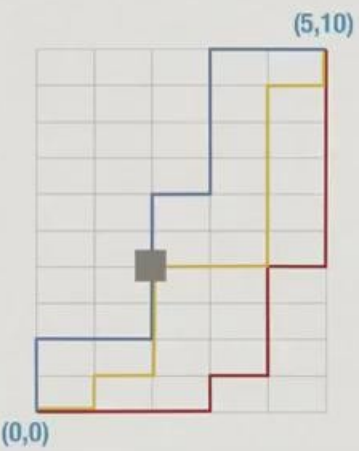
- What if an intersection is blocked?
- (2,4), for example
- Paths through (2,4) need to be discarded
- Two of our earlier examples are invalid paths

But the problem becomes more interesting, if we constrain it by saying that some of these intersections are blocked for instance, supposing there is some road work going on and we cannot go through this intersection (2, 4). This is the intersection 2 comma 4 second column and the fourth row counting from below. It's actually 2 comma 3, but 1, 2, 3, 4 yeah 2 comma 4. Now, if we cannot go through this then any path which goes through this particular block intersection should no longer be counted. Out to those 3003 some paths are no longer valid paths.

(Refer Slide Time: 04:49)

Holes

- What if an intersection is blocked?
- (2,4), for example
- Paths through (2,4) need to be discarded
- Two of our earlier examples are invalid paths




The diagram shows a grid with a black square at the intersection (2,4). Three paths are shown: a blue path that goes through the blocked intersection, a red path that avoids the block by going around it, and a yellow path that overlaps with the blue path in a section that is invalid due to the block.

For instance, in the earlier thing the blue path that we had drawn actually goes through this, the red path does not, where the yellow path overlapped with the blue path unfortunately in this bad section. It also passes through this. There are some paths which are allowed from the 3003 and some which are not. So, how do we determine how many paths survived this kind of block.

(Refer Slide Time: 05:09)

Combinatorial solution

- Every path through (2,4) goes from (0,0) to (2,4) and then from (2,4) to (5,10)
- Count these separately:
 - (4+2) choose 2 = 15
 - (6+3) choose 3 = 84
- Multiply to get all paths through (2,4): 1260
- Subtract from 15 choose 5 = 3003 to get valid paths that avoid (2,4): 1743



The diagram shows a grid with a black square at the intersection (2,4). A green square highlights the area around the block, with arrows indicating the paths from (0,0) to (2,4) and from (2,4) to (5,10).

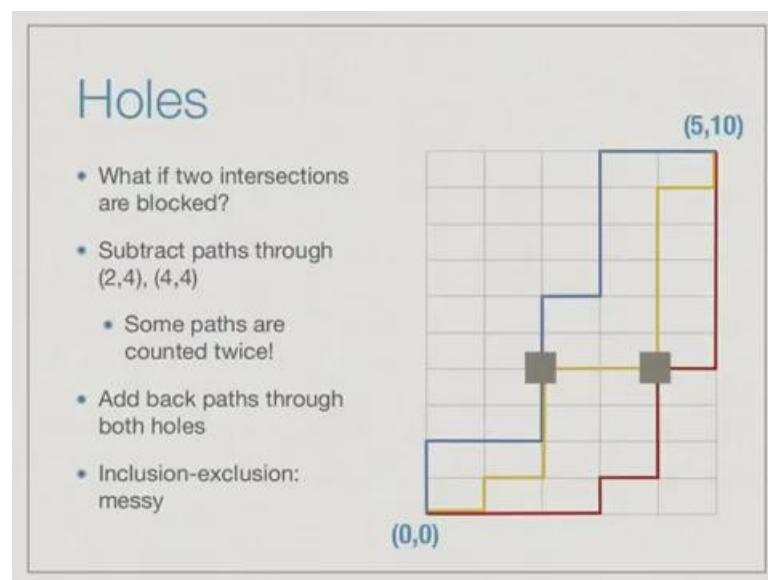
So, again we can use a combinatorial argument in order to be blocked a path must go to (2, 4) and then from (2, 4) to (5, 5). If we could only count how many paths go from (0,

0) to (2, 4) and then how many paths go from (2, 4) to (5, 10), these are all the bad paths. So, we can count these bad paths and subtract them from the good paths. How do we count the bad paths well we can just solve a smaller version of the problem. So, we have an intermediate target.

So, we solve this grid how many paths go from here to here, how many paths go from here to here. So, from (0, 0) to (2, 4) we get 4 plus 2 remember it 10 plus 5 it was a curve or get, 10; 4 plus 2 choose 2. So, we get 15 and from here to here the difference is that we have to do in both directions 3 and so, we have to go sorry we have to go up 6 and we have to go right 3, we are at (2, 4). So, we have to go from 4 to 10 and from 2 to 5.

So, we have 6 plus 3 choose 3, 84 ways of going from (2, 4) to this and each of the ways in the bottom, can be combined with a way on the top. So, we multiply this and we get 1260 paths which pass through this bad intersection, we subtract this from the original number 3003 and we get 1743 paths which remain. So, a combinatorial approach still works.

(Refer Slide Time: 06:32)

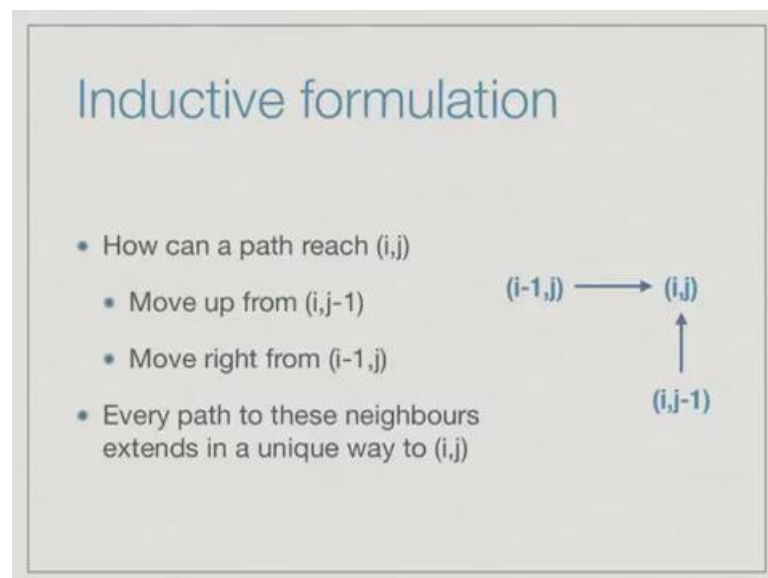


Now, what happens if we put 2 such intersections? So, we will you can do the same thing we can count all the parts which get blocked because of the first intersection, we can count all the paths which pass through in this case (4, 4) is the second intersection which has been blocked. So, we can count all these parts which pass through (4, 4). This we

know how to do: we just computed it for $(2, 4)$, but the problem is that there are some paths like the yellow paths which pass through both $(2, 4)$ and $(4, 4)$.

So, we need a third count we need to count paths which pass through both of these and make sure we do not double count them. So, one way is that we just add these back. This is something which is called in combinatorics inclusion and exclusion. So, when we have these overlapping exclusions, then we have to count the overlaps and include them back. We have to keep doing this step by step. If we have 3 holes we get an even more complicated inclusion exclusion formula and it rapidly becomes very complicated even to calculate the formula that we need to get. Is there a simpler way to do this?

(Refer Slide Time: 07:37)



Let us look at the inductive structure of the problem, suppose we say we want to get in one step to intersection (i, j) . How can we reach this in one step since our roads only go left to right and bottom to top, the only way we can reach (i, j) is by taking a right edge from its left neighbor. So, we can go from $(i-1, j)$ to (i, j) or we can go from below from $(i, j-1)$ to (i, j) . Notice that if a path comes from the left it must be different from a path that comes from below. So, every path that comes from the left is different from every path that comes from below. So, we can just add these up.

(Refer Slide Time: 08:20)

Inductive formulation

- $\text{Paths}(i,j)$: Number of paths from $(0,0)$ to (i,j)
- $\text{Paths}(i,j) = \text{Paths}(i-1,j) + \text{Paths}(i,j-1)$
- Boundary cases
 - $\text{Paths}(i,0) = \text{Paths}(i-1,0)$ # Bottom row
 - $\text{Paths}(0,j) = \text{Paths}(0,j-1)$ # Left column
 - $\text{Paths}(0,0) = 1$ # Base case

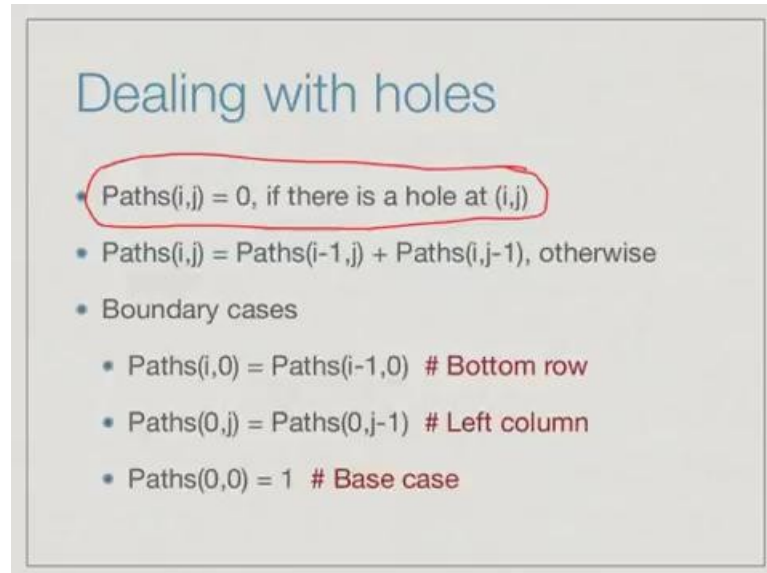
In other words if we say that $\text{paths}(i, j)$ is the quantity we want to compute, we want to count the number of paths from $(0, 0)$ to (i, j) . These paths must break up into 2 disjoint sets those which come from the left which recursively or inductively if you prefer to say is exactly the quantity $\text{paths}(i-1, j)$. How many paths are there which reach $(i-1, j)$ every one of these paths can be extended by a right edge to reach (i, j) and, they will all be different similarly $\text{paths}(i, j-1)$ are all those paths which come from below, because they all reach the point just below (i, j) from there each of them will be extended in a unique way to (i, j)

This gives us our simple inductive formula, $\text{paths}(i, j)$ is just the sum of $\text{paths}(i-1, j)$ and $\text{paths}(i, j-1)$. Then we need to of course, investigate the base cases: in this case the real base case is just $\text{paths}(0, 0)$: in how many ways can I go from $(0, 0)$ and just stay in $(0, 0)$? Well there is only one way, it is tempting to say 0 ways, but it is not 0 ways its one way otherwise nothing will happen. So, we have one way by just doing nothing to stay in $(0, 0)$ and if we are now moving along the left column, if you are moving along the left column then there are no paths coming from its left because we are already on the leftmost column.

So, all the paths to $(0,j)$ must be extensions of paths which have come from below up to $(0,j-1)$. Similarly if you are on the bottom row there is no way to come from below

because we are already on the lowest set of roads. So, $\text{paths}(i, 0)$ can only come from the left, from $\text{paths}(i-1, 0)$.

(Refer Slide Time: 09:56)



The slide is titled "Dealing with holes" in a blue font. It contains a bulleted list of recursive formulas for $\text{Paths}(i, j)$. The first bullet point, $\text{Paths}(i, j) = 0$, if there is a hole at (i, j) , is circled in red. The second bullet point is $\text{Paths}(i, j) = \text{Paths}(i-1, j) + \text{Paths}(i, j-1)$, otherwise. The third bullet point is "Boundary cases", which includes three sub-bullets: $\text{Paths}(i, 0) = \text{Paths}(i-1, 0)$ # Bottom row, $\text{Paths}(0, j) = \text{Paths}(0, j-1)$ # Left column, and $\text{Paths}(0, 0) = 1$ # Base case. The sub-bullets are indented and the labels "# Bottom row", "# Left column", and "# Base case" are in red.


Dealing with holes

- $\text{Paths}(i, j) = 0$, if there is a hole at (i, j)
- $\text{Paths}(i, j) = \text{Paths}(i-1, j) + \text{Paths}(i, j-1)$, otherwise
- Boundary cases
 - $\text{Paths}(i, 0) = \text{Paths}(i-1, 0)$ # Bottom row
 - $\text{Paths}(0, j) = \text{Paths}(0, j-1)$ # Left column
 - $\text{Paths}(0, 0) = 1$ # Base case

This gives us a direct way to actually compute this even with holes because, the only difference now is that if, there is a hole we just declare that no paths can reach that place. So, we just add an extra clause which says $\text{paths}(i, j)$ is 0 if there is a hole at (i, j) ; otherwise we use exactly the same inductive formulation and now what happens is, if I have a hole below me, if I have a hole below me, no paths can come from that direction because by definition $\text{paths}(i, j)$ at that point is 0.

(Refer Slide Time: 10:29)

Computing Paths(i,j)



- Naive recursion will recompute multiple times
- Paths(5,10) requires Paths(4,10) and Paths(5,9)
- Both Paths(4,10) and Paths(5,9) require Paths(4,9)
- Use memoization ...
- ... or compute the subproblems directly in a suitable way

So, once again if we now apply this and do this using the standard translation from the inductive definition to a recursive program, we will find that we will wastefully recompute the same quantity multiple times for instance $\text{paths}(5,10)$. If we have $\text{paths}(5,10)$, it will require me to compute this and this.

These are the 2 sub problems for $\text{paths}(5,10)$, namely (4, 10) and (5, 9) but, in turn in order to compute (4, 10) I will have to compute whatever is to its left and below it and in order to compute (5, 9) I will also have to compute what is to its left and below it and now what we find is that this quantity namely (4, 9) is computed twice, once because of the left neighbor of (5, 10) and once because of the neighbor below (5, 10).

So, as we saw before we could use memoization to make sure that we never compute (i,j) twice by storing a table i comma j , and every time we compute a new value for i comma j we store it in the table and every time we look up, we need to compute one we first check the table, if it is already there we look it up, otherwise we will compute it and store it, but since we know there is a table and we know what the table structure is basically it is all entries of the form i comma j . We can also see if we can fill up this table iteratively by just examining the sub problems in terms of their dependencies.

(Refer Slide Time: 12:01)

Dynamic programming

(5,10)

- Identify dependency structure
- Paths(0,0) has no dependencies
- Start at (0,0)

In general a node the value depends on things to its left and below. If there are no dependencies, it must have nothing to its left and nothing below and there is only one such point namely (0, 0). This is the only point which is the base case which has nothing to its left and nothing below so its value is directly read. So, we start from here.

(Refer Slide Time: 12:24)

Dynamic programming

- Start at (0,0)
- Fill row by row

1	11	51	181	526	1363
1	10	40	130	345	837
1	9	30	90	215	492
1	8	21	60	125	272
1	7	13	39	65	147
1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

Remember that the base value at (0, 0) is one, and now once we have done this it turns out: you remember the road dependency, it said (i, 0) is (i-1, 0). So, we can fill up this, because this has only one dependency which is known now. In this way I can fill up the

entire row and say that all along this row there is only one path namely the path that starts going right and keeps going right. Now we can go up and see that this thing is also known because, it also depends only on the value below it and once that is known then these 2 are known.

So I can add them up; remember the value at any position is just the value to its left plus the value to its bottom and now I start to get some non trivial values, and in this way I can fill up this table row by row and at each point when I come to something I will get the fact with the dependency unknown. The next row looks like this and the next row. Now we come to the row with holes. So, for the row with holes, wherever we hit a hole instead of writing the value that we would normally get by adding its left and bottom neighbour we deliberately put a 0 because; that means, that no path is actually allowed propagating through that row.

Now, when we come to the next row, the holes will automatically block the paths coming from the wrong direction. So, here for instance we have only 6 paths coming from the left because we have no paths coming from below similarly we have 26 paths coming from the left and no paths coming from below. This is how our inductive definition neatly allows us to deal with holes and from that inductive definition we recognize the dependency structure and we imagine the memo table and now we are filling up this memo table row by row so that at every point when we reach an (i, j) value its dependent values are already known.

So, we can continue doing this row by row, and eventually we find look there are 1363 paths which avoid these two.

(Refer Slide Time: 14:18)

Dynamic programming						
1	11	51	181	526	1363	
1	10	40	130	345	837	
1	9	30	90	215	492	
1	8	21	60	125	272	
1	7	13	39	65	147	
1	6	6	26	26	82	
1	5	0	20	0	56	
1	4	10	20	35	56	
1	3	6	10	15	21	
1	2	3	4	5	6	
1	1	1	1	1	1	

So, we could also do the same thing in a different way instead of doing the bottom row, we can do the left column and the same logic says, that we can go all the way up then we can start in the second column, go all the way up and do this column by column and not unexpectedly, we should get the same answer. There is a third way to do this.

(Refer Slide Time: 14:39)

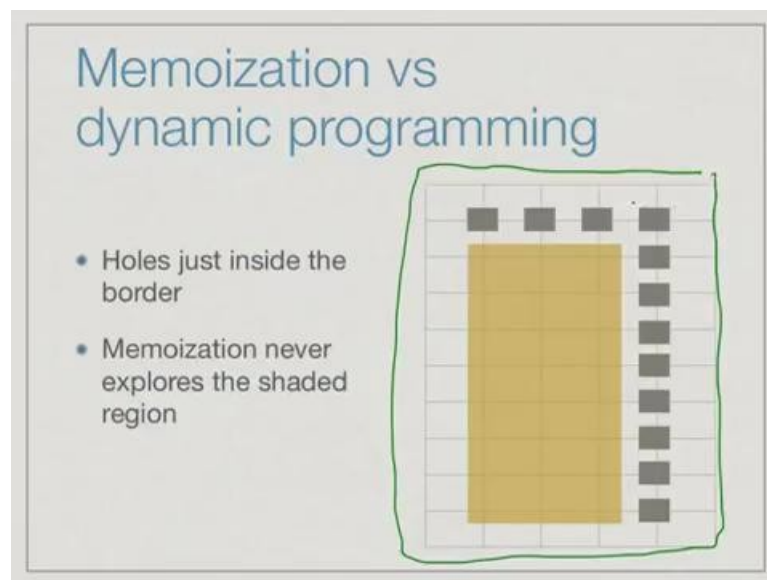
Dynamic programming						
1	11	51	181	526	1363	
1	10	40	130	345	837	
1	9	30	90	215	492	
1	8	21	60	125	272	
1	7	13	39	65	147	
1	6	6	26	26	82	
1	5	0	20	0	56	
1	4	10	20	35	56	
1	3	6	10	15	21	
1	2	3	4	5	6	
1	1	1	1	1	1	

So, once we have one at (0, 0) then we can fill both the first element above it and the first element to its right. So, we can do this diagonal, now notice that any diagonal value like this one has both its entries. This has only one entry, this also. So I can now fill up this

diagonal. I can go one more diagonal, then I can go one more diagonal. So, we can also fill up this thing diagonal by diagonal.

The dependency structure may not require us to fill it in a particular way we might have very different ways to fill it up, all we want to do is systematically fill up this table in an iterative fashion not recursively we do not want to call f of i, j and then look at f of i minus 1, j . We want to directly say when we reach (i, j) we have the values we need, but the values we need could come in multiple different orders. So, we could have done it row wise, we could have done it column wise and here you see we can do it diagonally, but it does not matter so long as we actually get all the values that we need.

(Refer Slide Time: 15:36)



So, one small point, so we have said that we can use Memoization or we can use dynamic programming. One of the advantages of using dynamic programming is it avoids this recursive call. So, recursion we had mentioned earlier, also in some earlier lecture, comes with a price because whenever you make a recursive call, you have to suspend a computation, store some values, restore those values. There is a kind of administrative cost with recursion.

So, actually though it looks like only a single operation and we call fib of n minus 1 or fib of n minus 2. There is actually a cost involved with suspending this operation, going there and coming back. So, saving on recursion is one important reason to move from Memoization to dynamic programming, but what dynamic programming does is to

evaluate every value regardless of whether its going to be useful for the final answer or not.

In the grid path thing there is one situation where you can illustrate this. Imagine that we have these obstacles placed exactly one step inside the boundary. Now, if we want to reach this its very clear that I can only come all the way along the top row or all the way up the rightmost column, there is no other way I can reach them. So, anything which is inside this these positions there is no way to go from here out. There is no point in counting all these values.

We have this region which is in the shadow of these obstacles which can never reach the final thing. So, when we do memoization when we come back and recursively explore it will never ask us to come here because it will never pass these boundaries. On the other hand our dynamic programming will blindly walk through everything. So, it will do row by row, column by column and it will eventually find the 0s, but it will fill the entire n by n grid. In this case how many will memoization do? It will do basically only the boundary. It will do only order $m+n$.

(Refer Slide Time: 17:32)

Memoization vs dynamic programming

- Memo table has $O(m+n)$ entries
- Dynamic programming blindly fills all $O(mn)$ entries
- Iteration vs recursion — “wasteful”
dynamic programming is still better, in general

The diagram shows a 10x10 grid. A yellow shaded region covers the first 8 rows and the first 6 columns. To the right of this region, in the 9th and 10th columns, there are several black squares representing obstacles. The yellow region represents the area explored by dynamic programming, while the black squares represent obstacles that define the boundary for memoization.

So, we have a memo table which has only a linear number of entries in terms of the rows and columns and a dynamic programming entry, which is quadratic; if both were n it will be n squared, thus is $2n$. This suggests that dynamic programming in this case, is wastefully computing a vast number of entries. So n squared is much larger than $2n$

remember. It will take us enormous amount of time to compute it, if we just count the cost per entry, but the flip side is that each entry that we need to add to the memo table requires one recursive call.

The reality is that these recursive calls will typically cost you much more, than the wastefulness of computing the entire table. In general even though you can analyze the problem and decide that memoization will result in many fewer new values being computed than dynamic programming. It is usually sound to just use dynamic programming as the default way to do the computation.