LET'S GO TO WORK

CSE 331

Instructor Sebnem Onsay

Dynamic Programming problem


DP

Problem Definition

You are provided with 2 positive integers : **width** and **height** of a grid shaped , rectangular graph.

Your job is to write a python method that will return the number of ways to arrive the bottom right corner of this graph.
Starting point will be the top left corner.
Your moves are limited, you can either go down or right.
You can never go up or left in the graph.

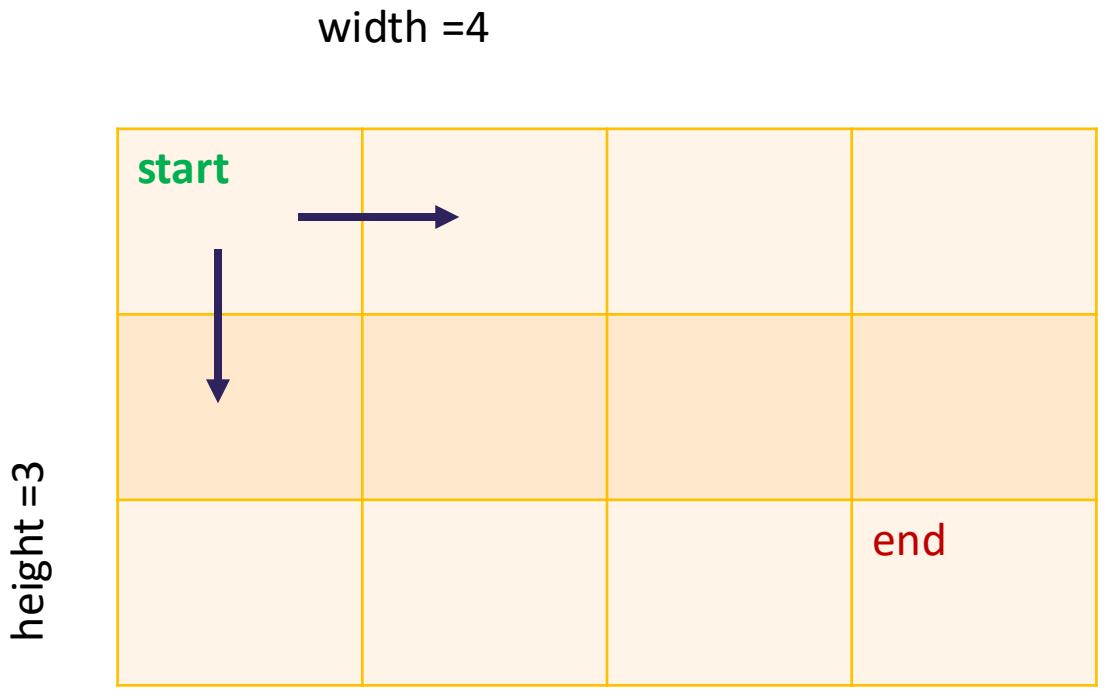
Run Time O(n*m), where n is the width, m is the height

Space : O(n*m)

You can do better than this run time but the point of this exercise you to practice with DP, using a grid

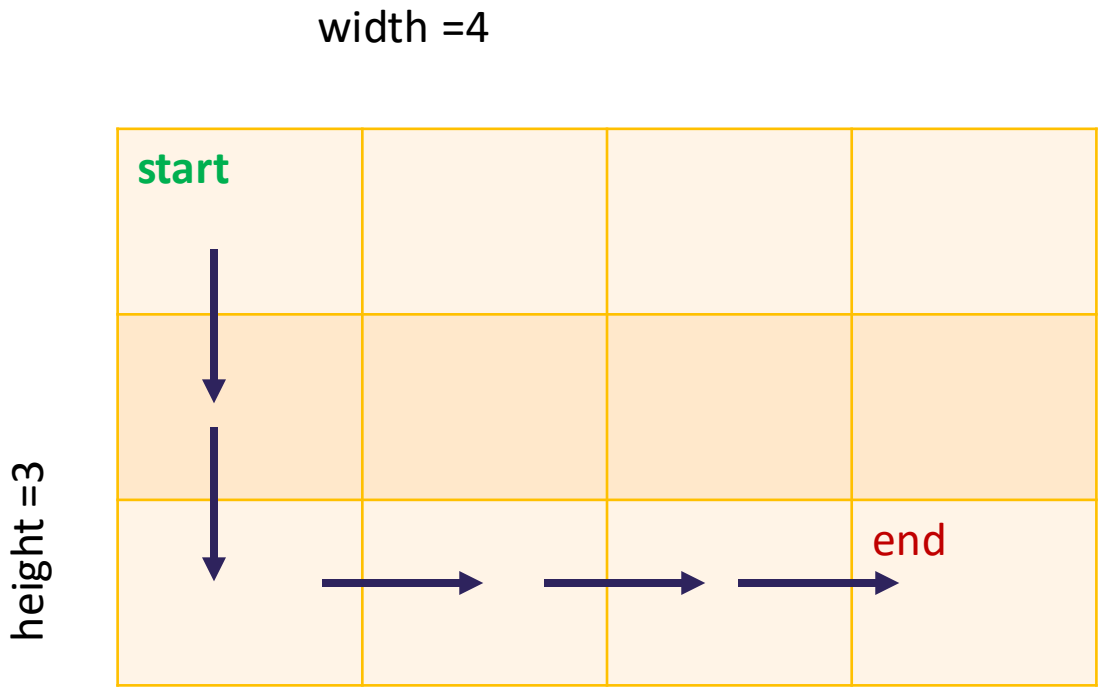
For Example

- Given the graph as shown below, width =4 , height =3
- Note that you may assume width * height >=2 , you will never be given 1 X 1 grid



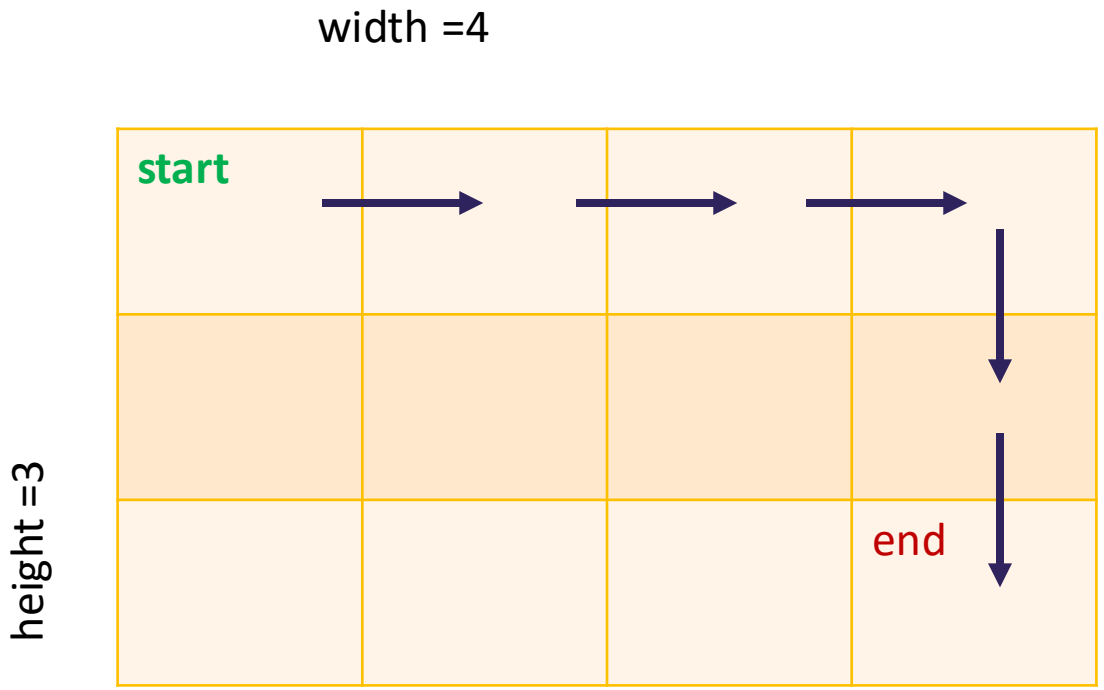
You can either go down or right

For Example



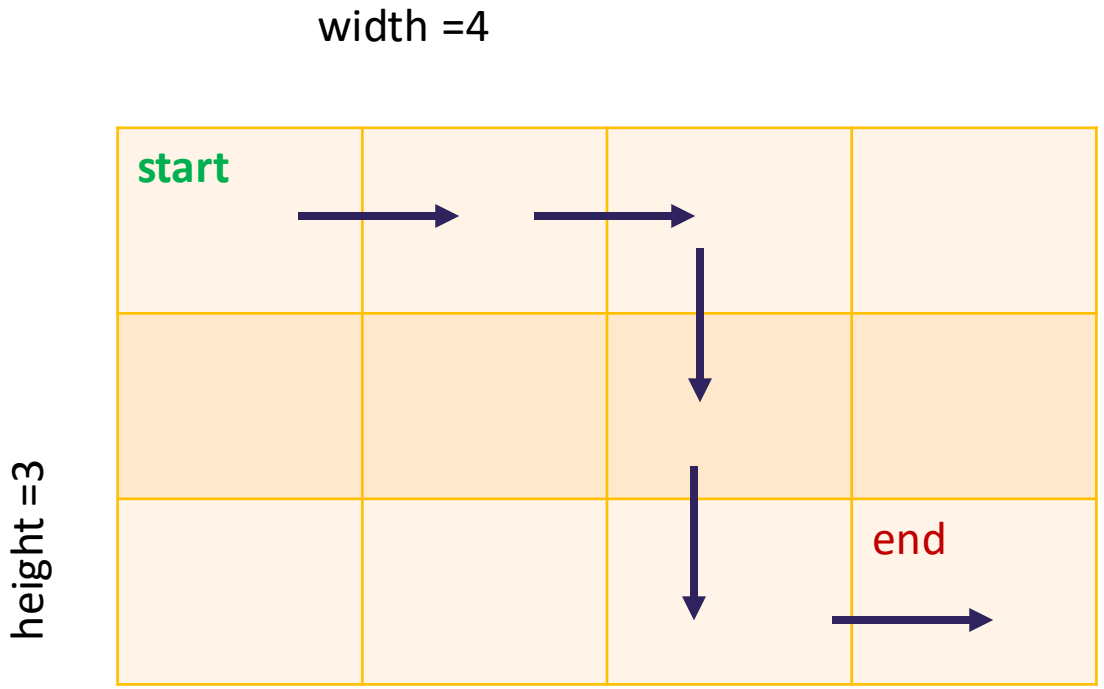
You can either go down or right

For Example



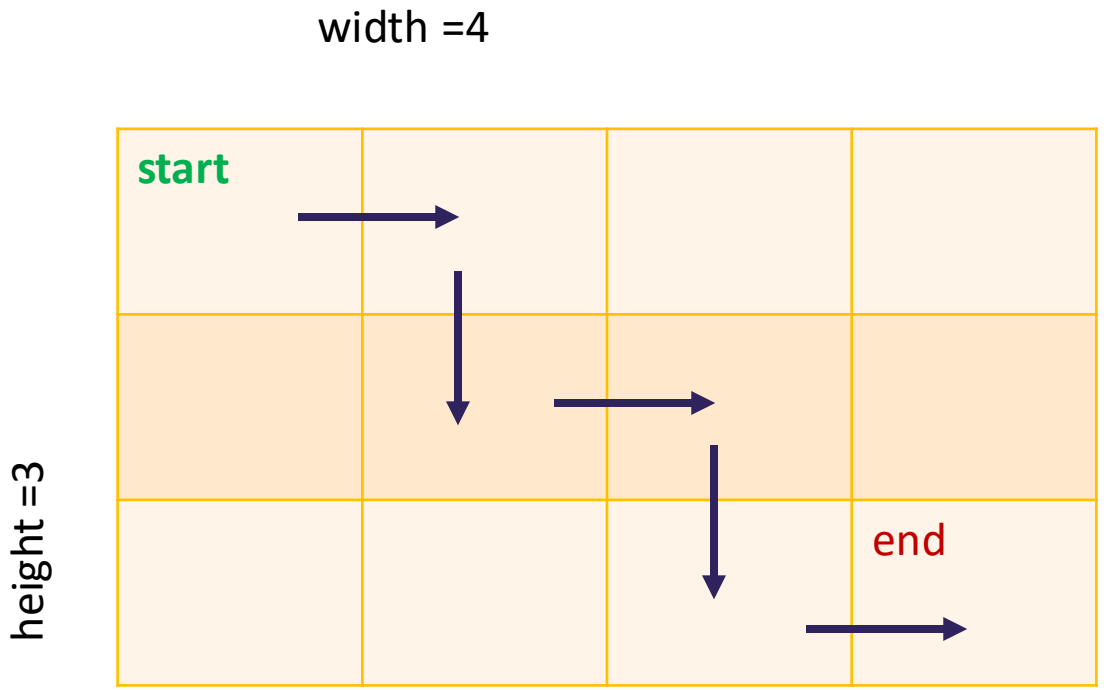
You can either go down or right

For Example



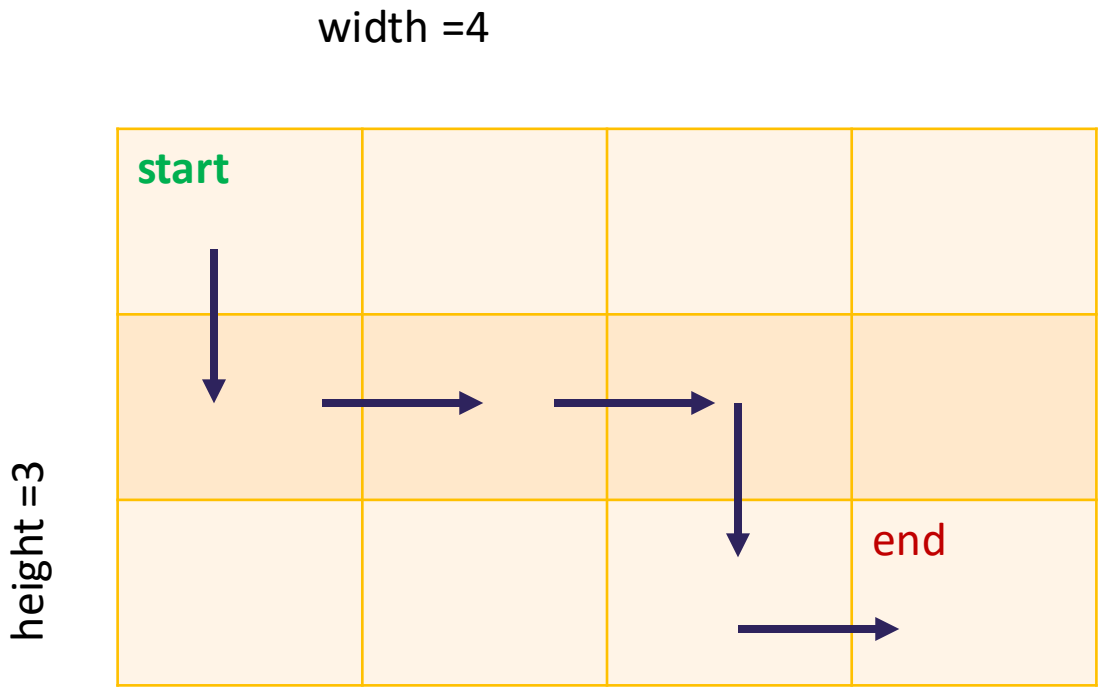
You can either go down or right

For Example



You can either go down or right

For Example



You can either go down or right

Hint 1

- You can think recursively.
  - How many positions in the graph can access the bottom right corner of the graph?
    - In other words , what positions do you need to reach before you can reach the bottom right corner?

Hint 2:

- Number of ways to reach any position in the graph =no_of ways to reach position directly above + no_of ways to reach position directly to its left.
- This is because you have limited moves recall that can only travel down and right.

Hint 3:

- Using the information in Hints 1 and 2, can you produce an efficient way to solve this problem that does not repeatedly perform the same work? What does a dynamic programming implementation look like?

Hint 4:

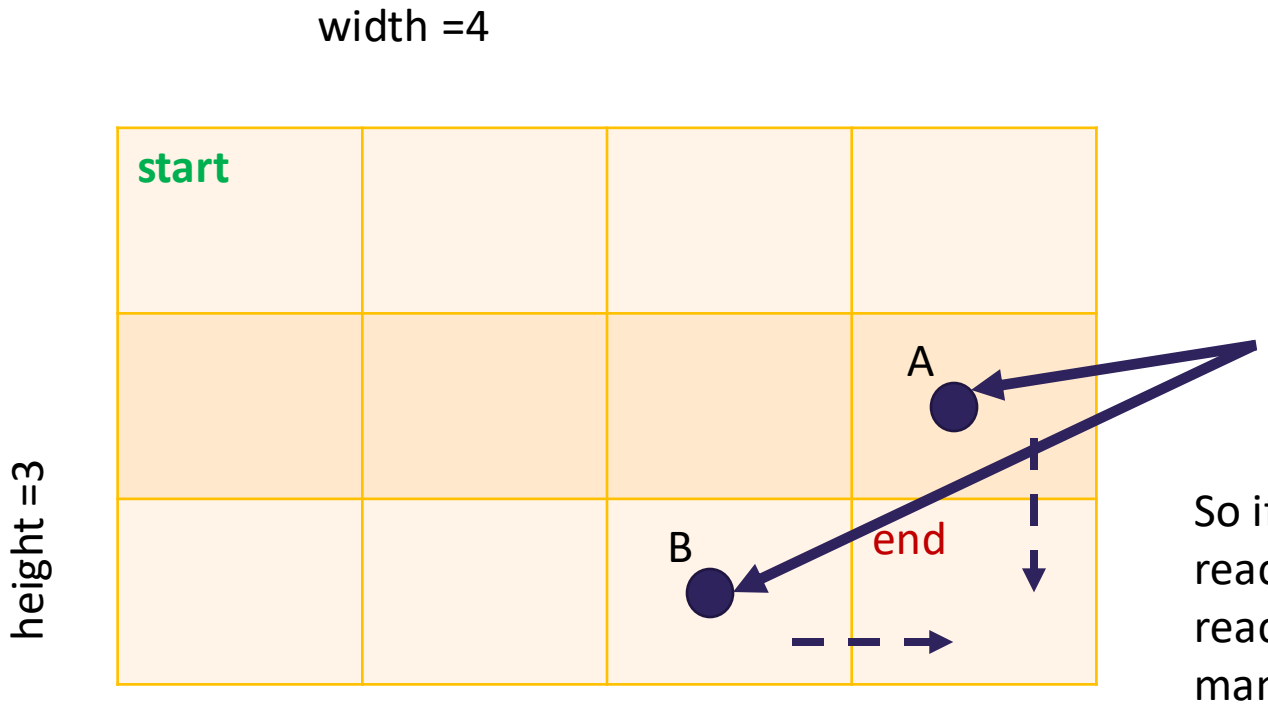
- To efficiently solve this problem, simply loop through the entire graph, column by column, row by row, and calculate the number of ways to reach each position.
- If you are on the top or left edge of the graph, there is only one way to reach your position. If you are anywhere else in the graph, number of ways to reach your position is the number of ways to reach the position directly above it plus the number of ways to reach position directly to its left (which you already calculated and should be storing).
- Every time you calculate the number of ways to reach a position, store the answer so that you can use it later in the calculation of other positions.

Optimal Time and Space Complexity

- Run Time O(n+m), where n is the width, m is the height
- Space : O(1)
- But this can be a tricky to achieve it involves using permutations. And usually if this is a question in an interview, you are not expected to solve this using a mathematical formula that will involve permutation.

Solving it with recursion

- We can only move down and we can only move right…

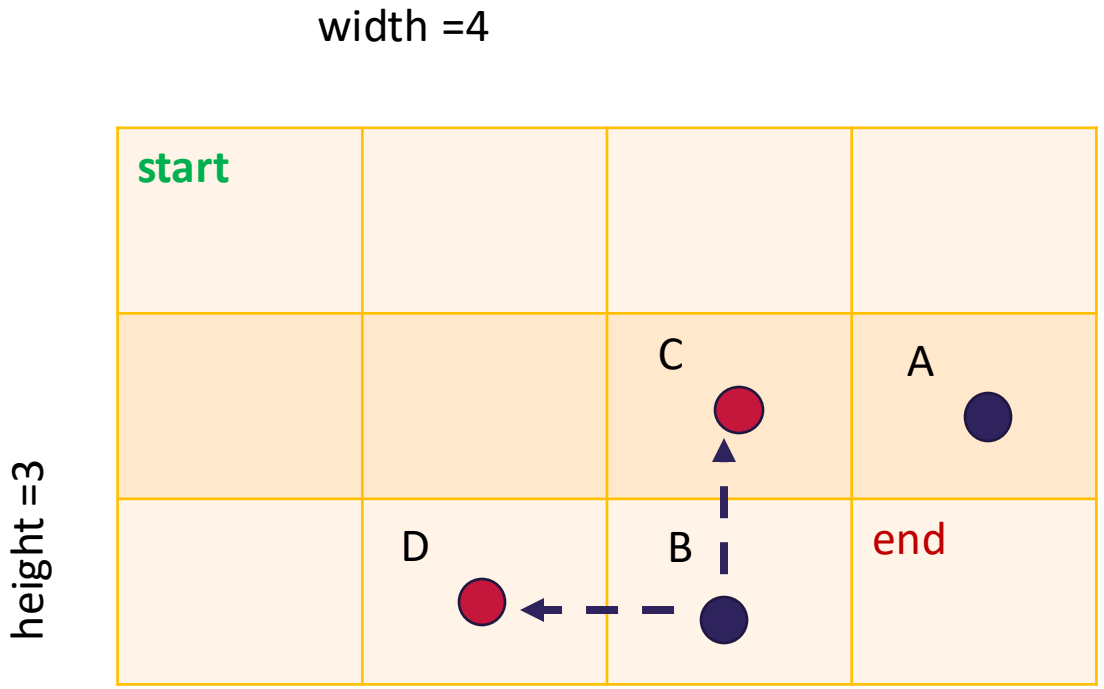


We need to reach to square A or Square B one to get to end point !

So if we know number of ways there is to reach to square A or number of ways to reach to square B, we will know how many ways to reach to end, it is the sum of the ways to reach Square A + Square B

Solving it with recursion

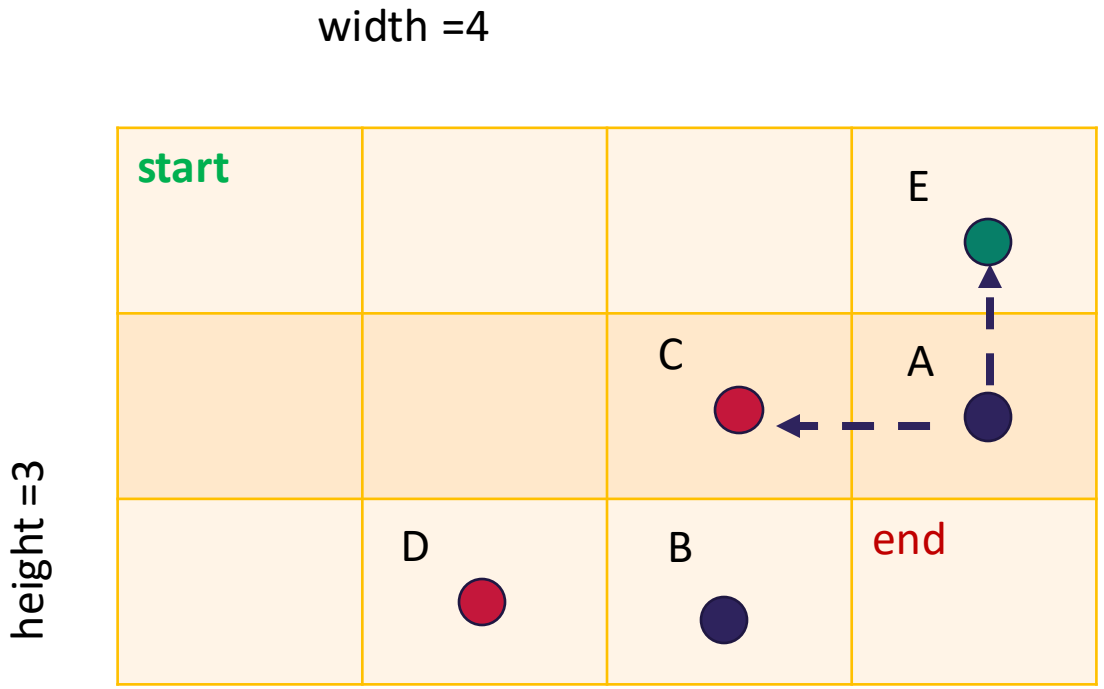
- Now our goal is the find the number of ways to get to Square A and Square B. Lets start with Square B here



Number of ways to get to Square B?
the sum of Square C + Square D will help us with number of ways to get to Square B

Solving it with recursion

- How do we get to Square A?



Number of ways to get to Square A ?

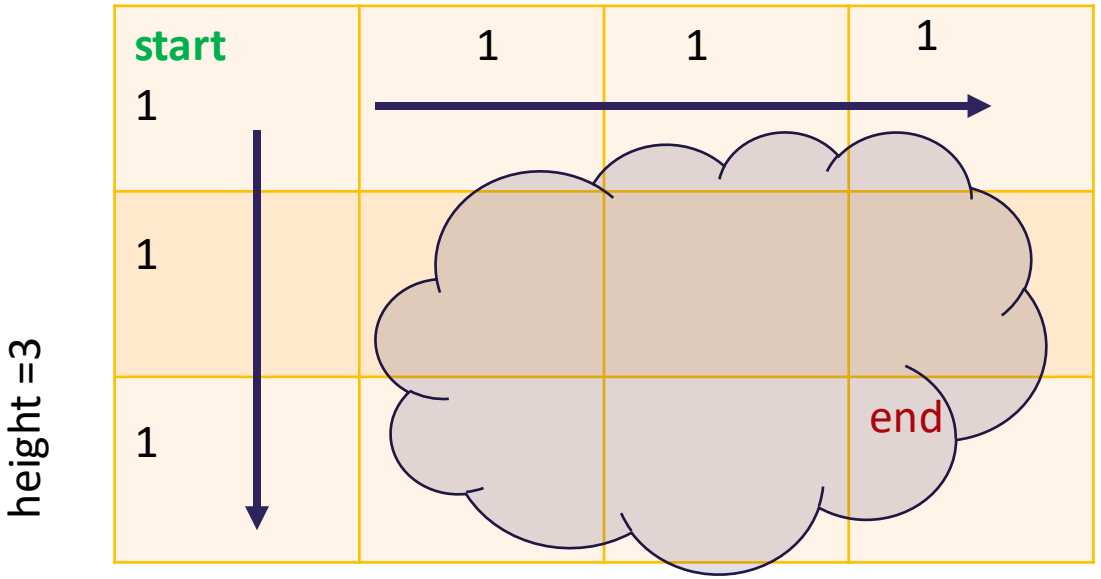
The sum of Square C + Square E will help us get the number of ways to get to Square A

This is the general idea for the recursive solution, we are going to start at the very bottom…at the end square

Recursive Solution

width =4

We can only get to these squares on the **border** with 1 way.
There is exactly one way to get these square

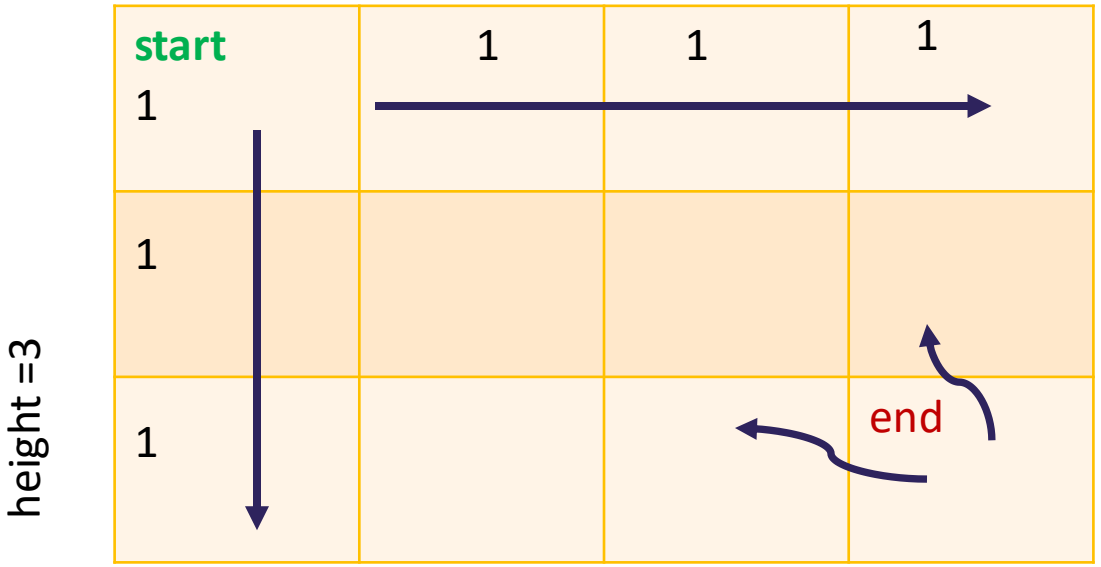


But we do not know how to get to these squares, so we start from the end

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square



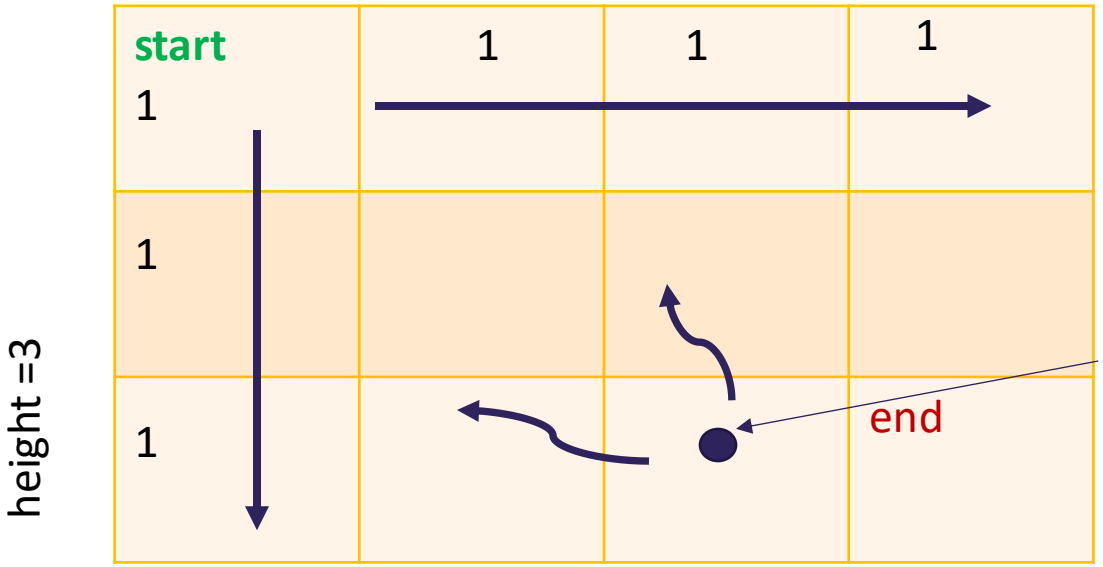
But we do not know how to get to these squares, so we start from the end

So let's start with the **left of the end** first.

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square



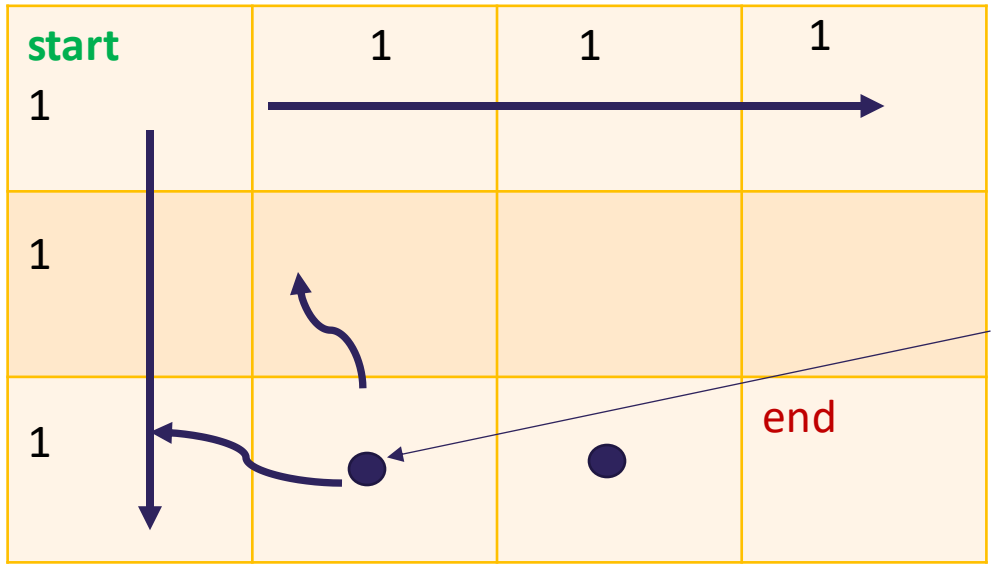
How did we get here, let's pick the left one again

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

height =3



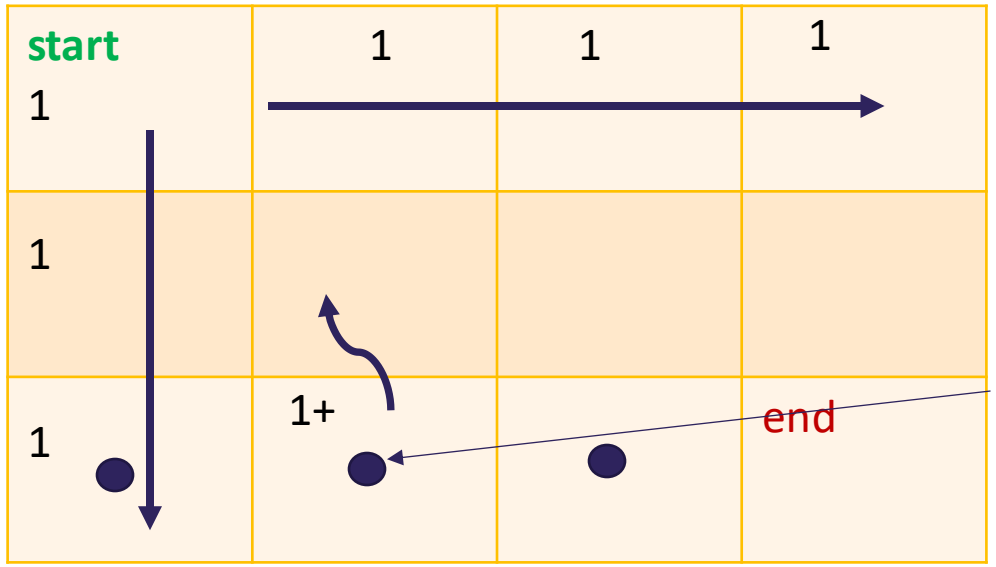
How did we get here, go to left again

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

height =3



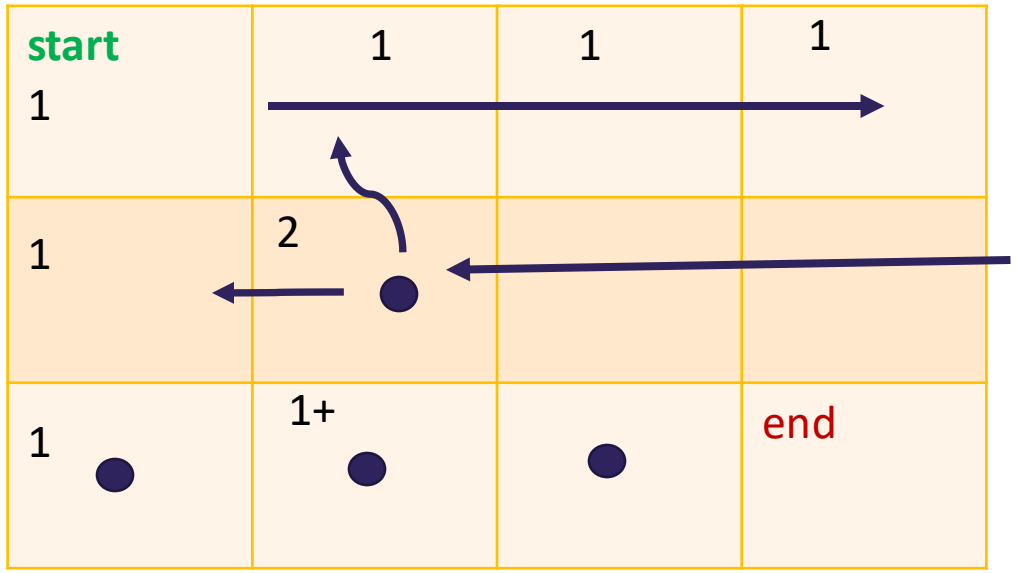
We will see that is the border, we will mark it +1 and now look up

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

height =3

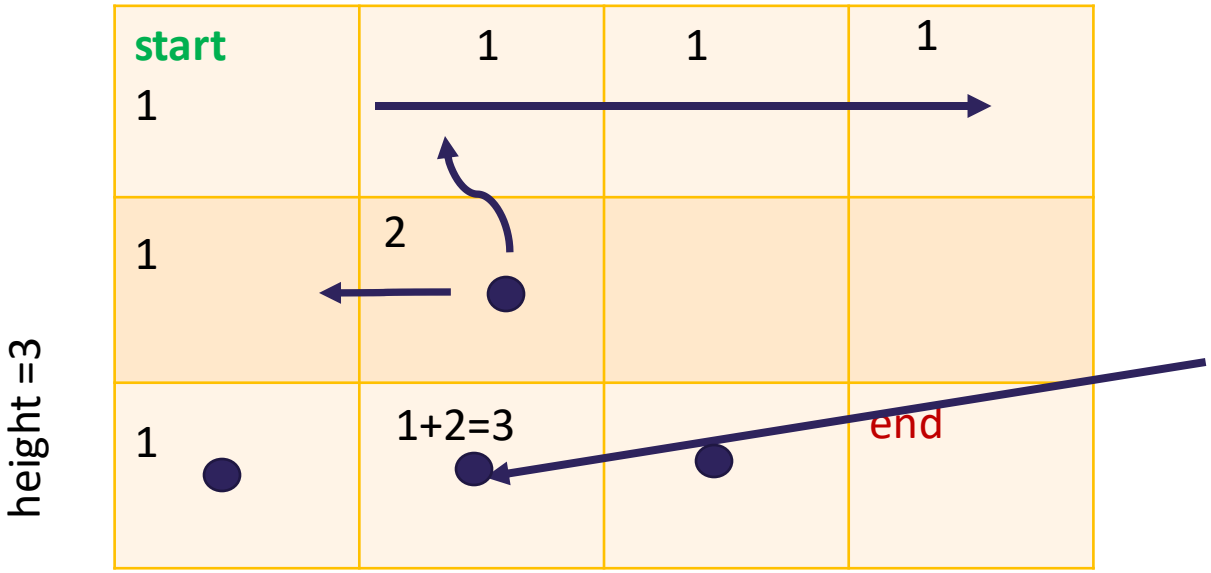


We see that there are 2 ways to get to this square

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

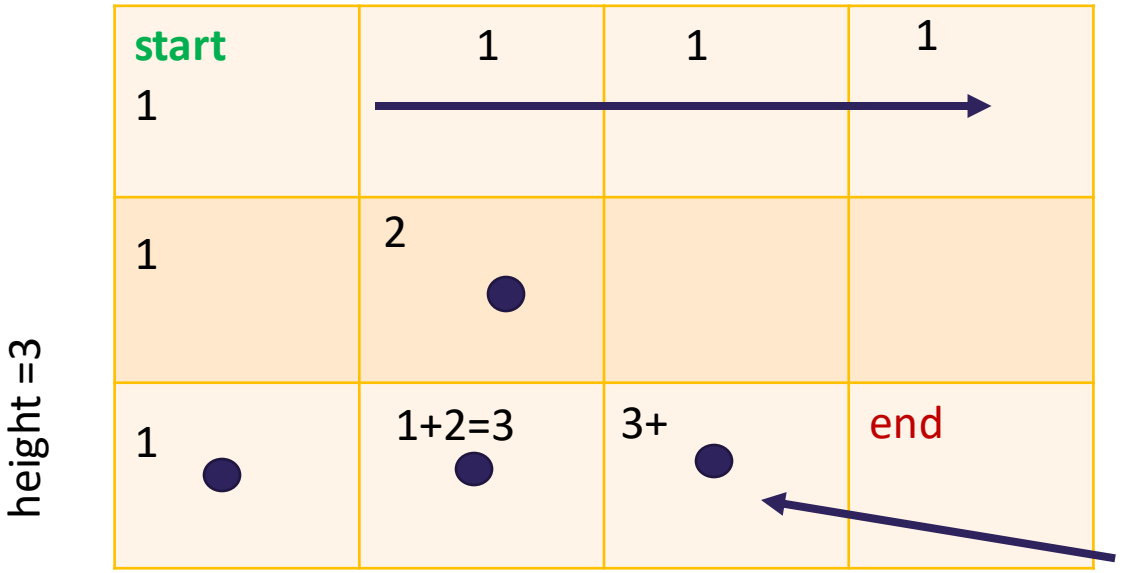


Now we can add +2 here, which means we can get to this square in 3 different ways, so this 3 will be returned back to where it was called from .

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

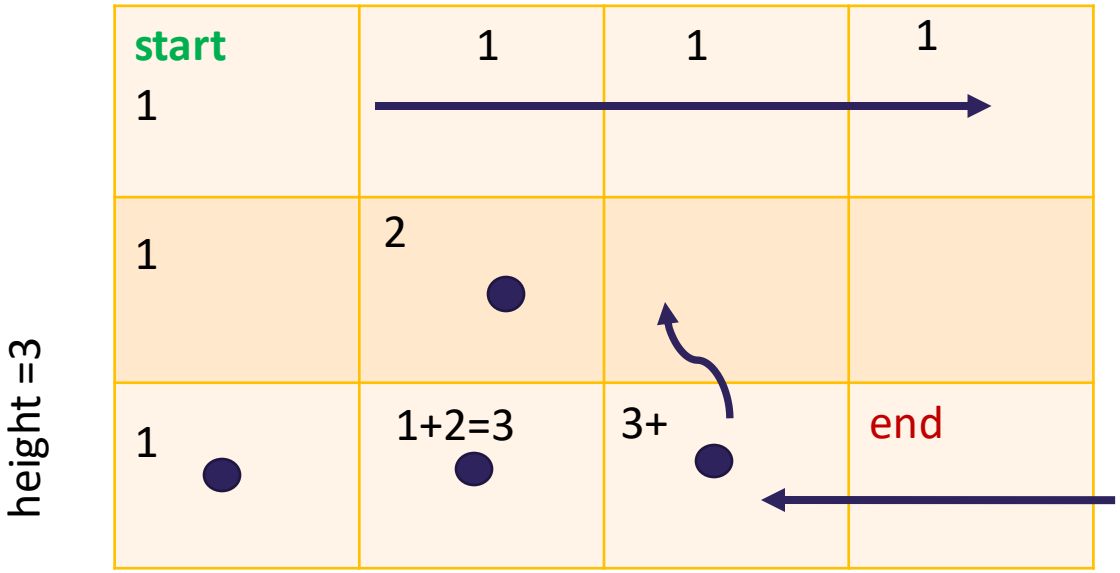


Now we can add +3 here, which means we can get to this square in 3 different ways, so this 3 will be returned back to where it was called from

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

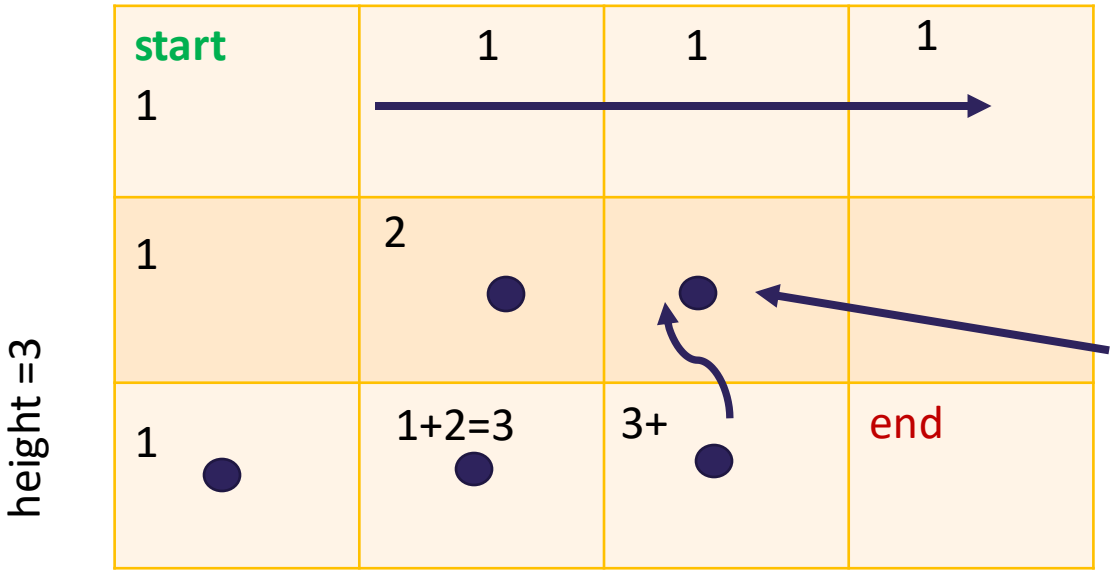


Now this square needs to look up..
We just completed the left …

Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

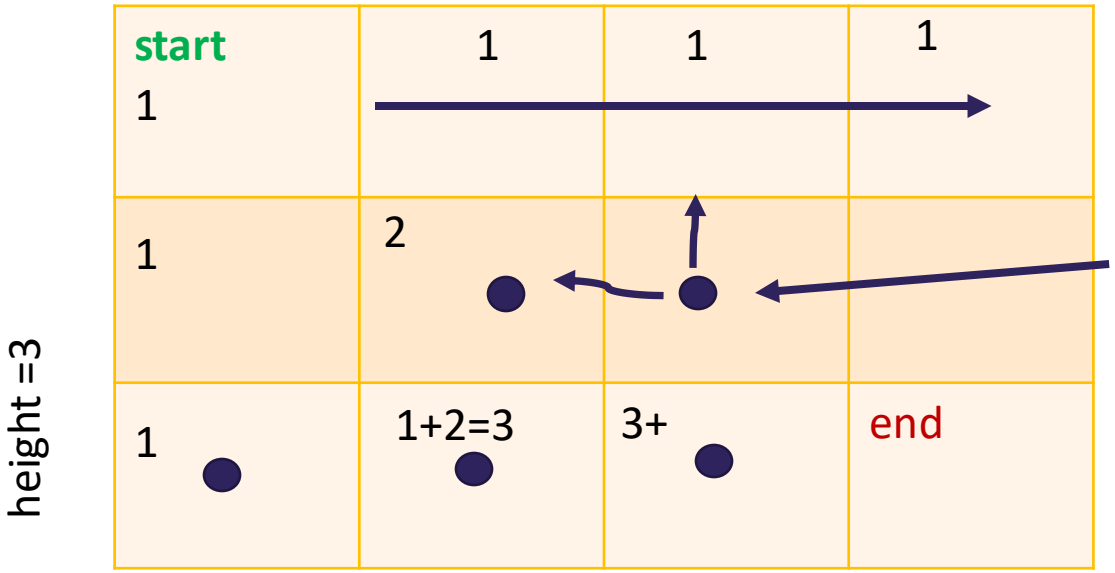


How many number of ways to get here? We do not know, so we have to look left and we have to look up. Let's look left first

Recursive Solution

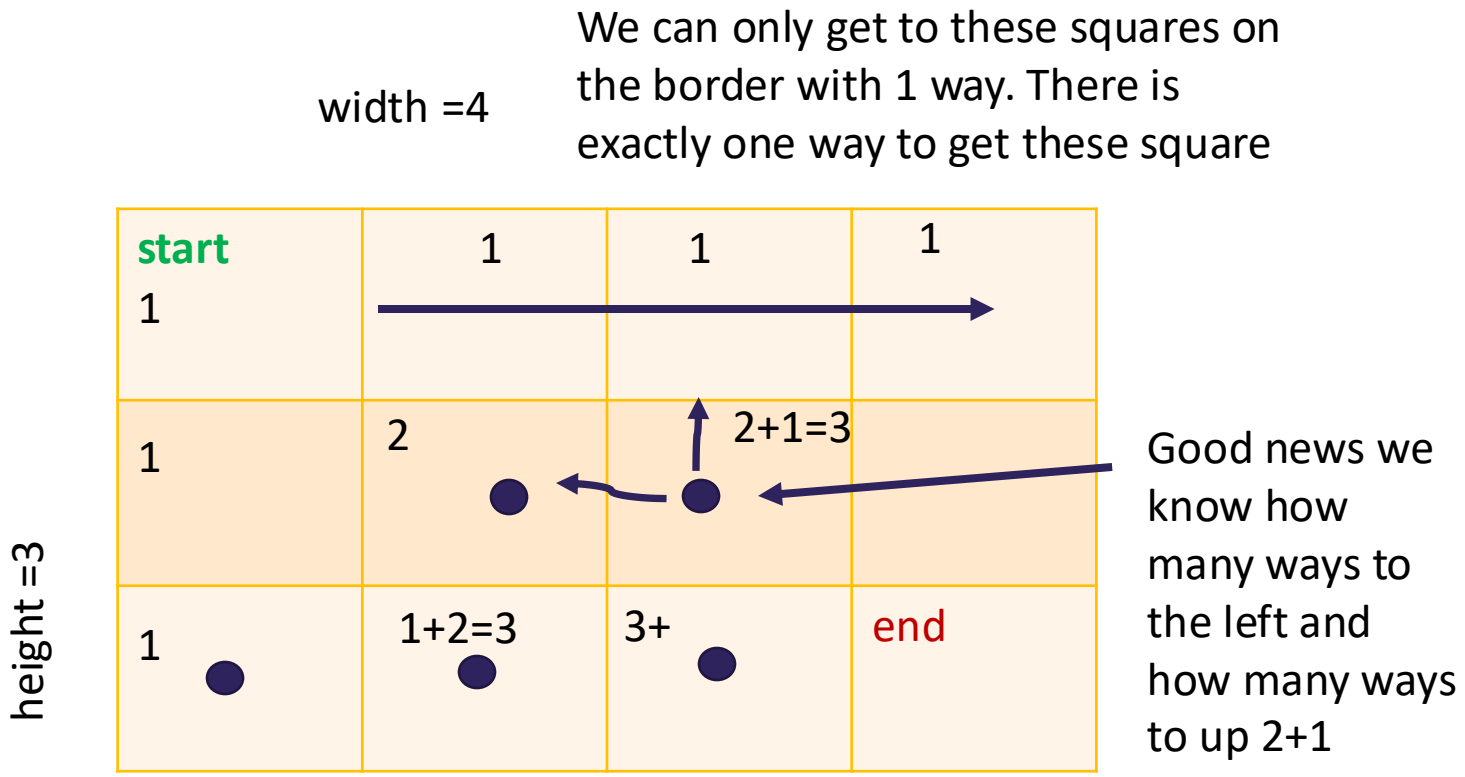
width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

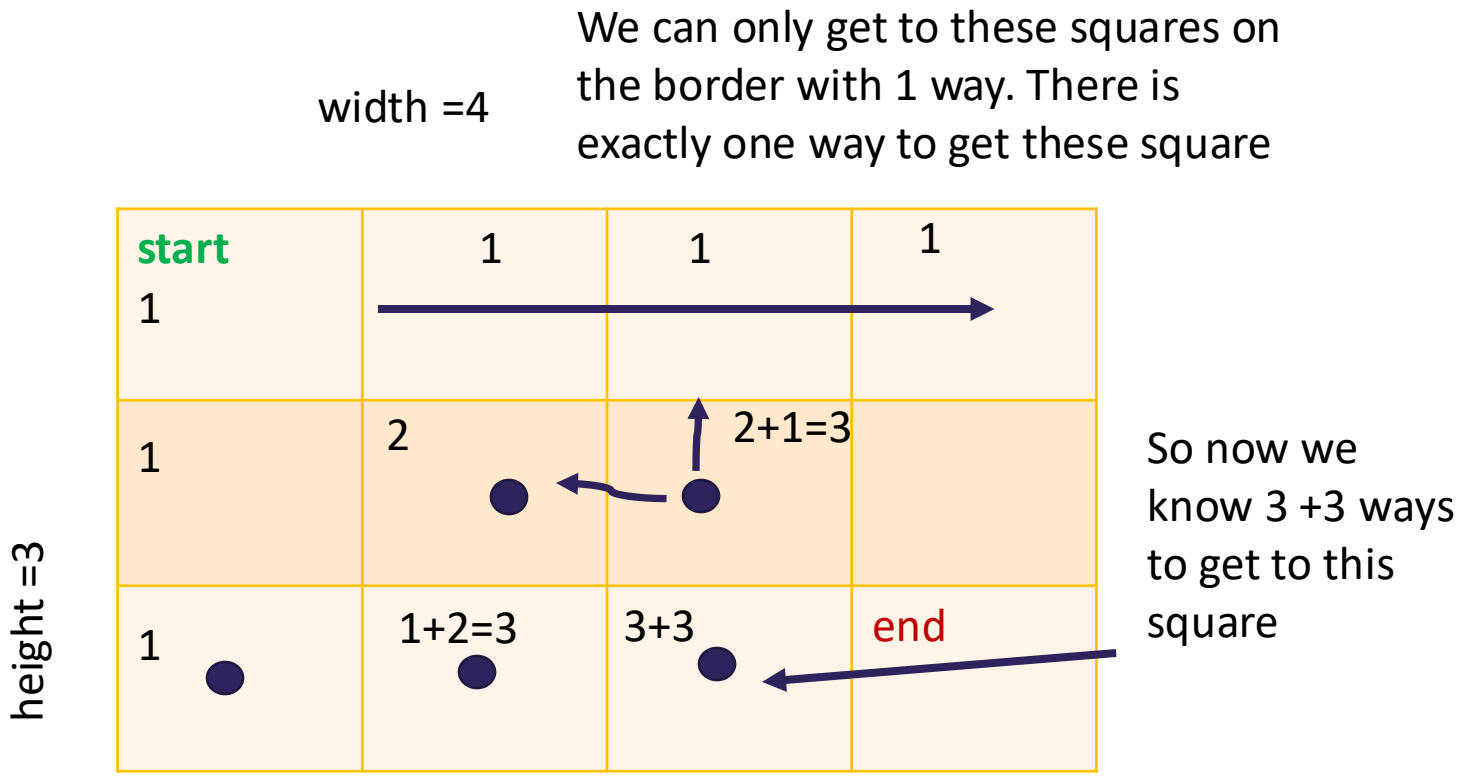


Good news we know how many ways to the left and how many ways to up 2+1

Recursive Solution



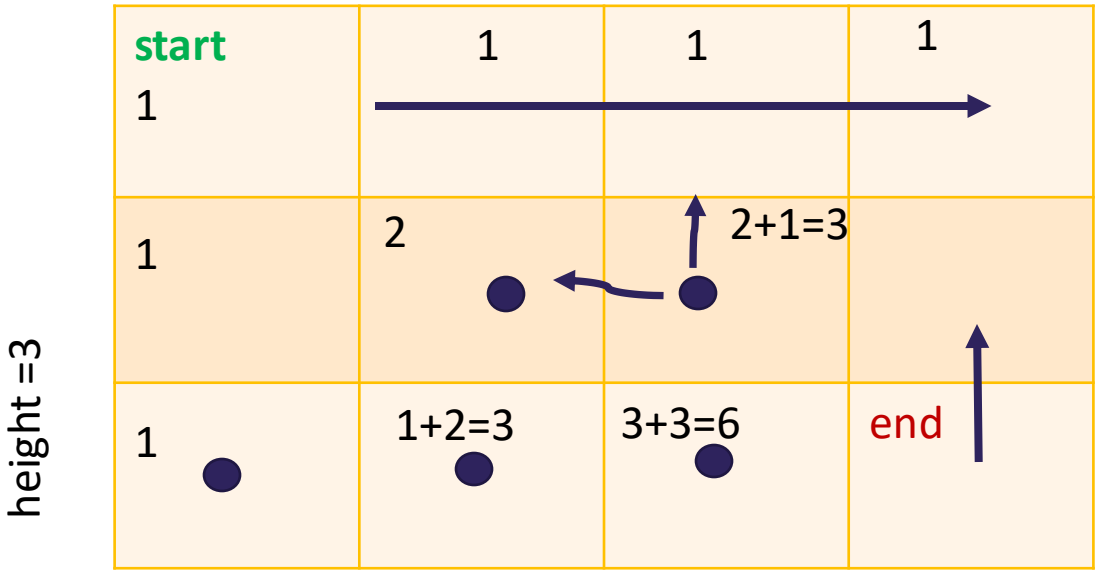
Recursive Solution



Recursive Solution

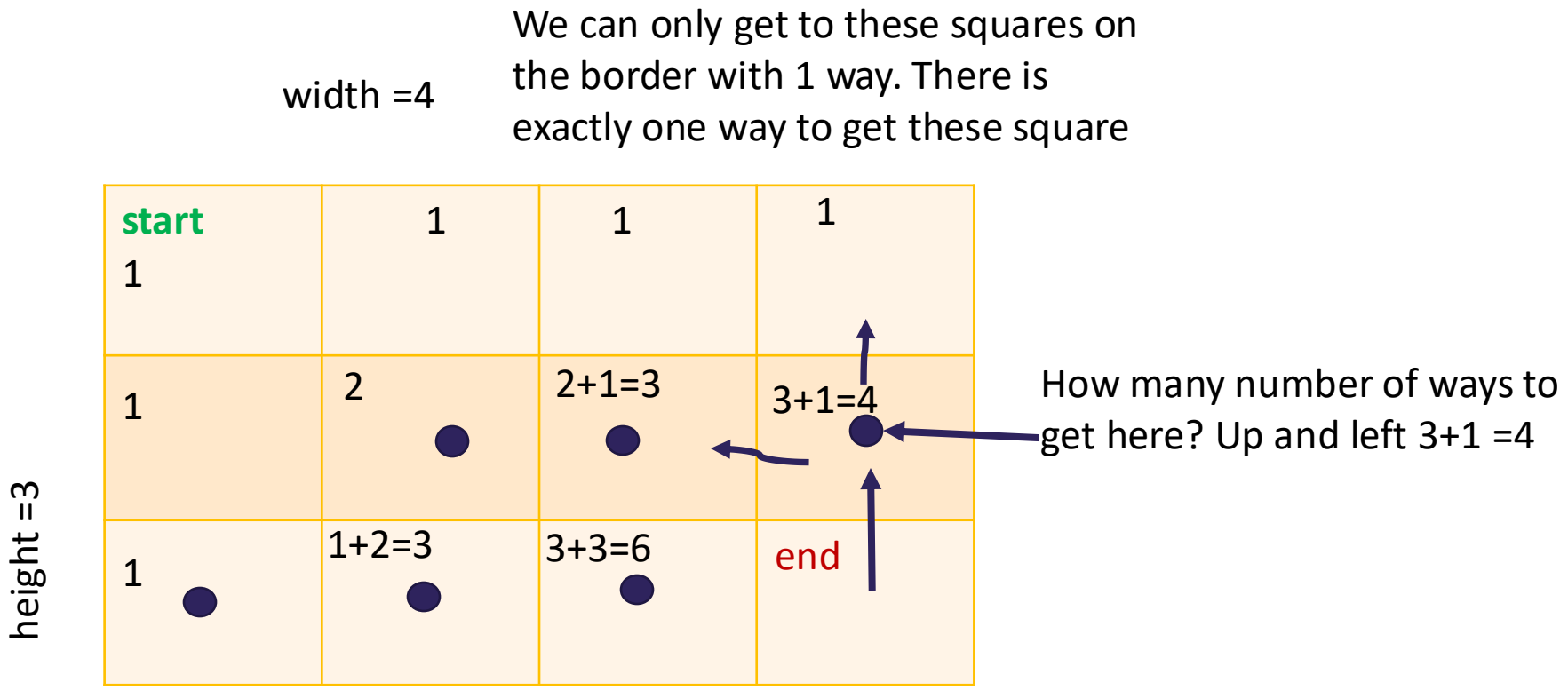
width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square



We finished going left from the end square now we need to go up…

Recursive Solution

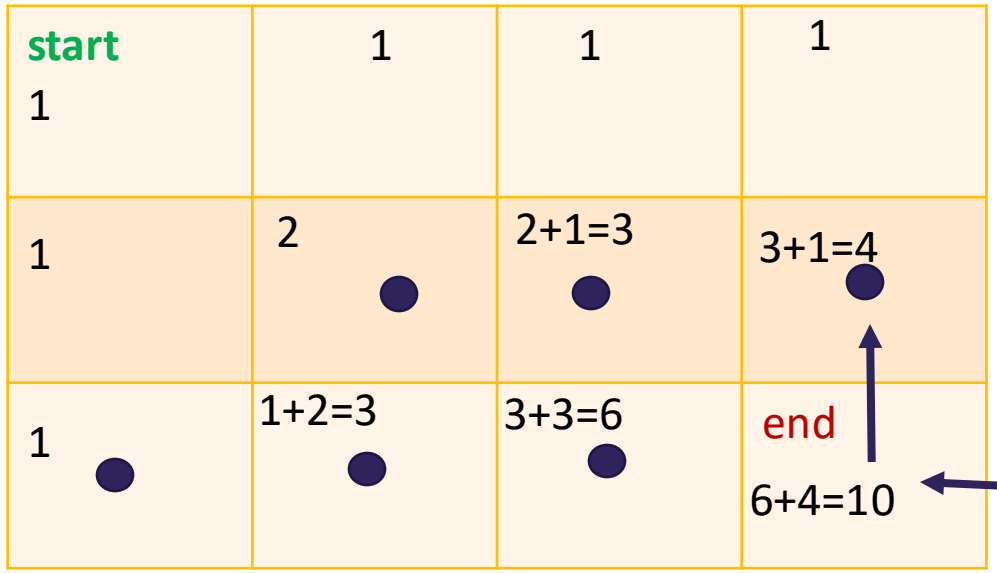


Recursive Solution

width =4

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

height =3



How many number of ways to get here? Up and left **6+4 =10**

Our recursive approach vs what we just talked about

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

width =4

height =3

start 1	1	1 G	1 E
1	2 F	2+1=3 C	3+1=4
1	1+2=3	3+3=6	end 6+4=10

So in our recursive approach we are not really storing the value of how many ways we actually got to Square C and how many ways we got to Square E?

In order for us to know how many ways we got to Square C, we need to look up and we need to look left.
Looking left is Square F
We are not really storing that value, instead we still need to check how many ways we got to Square F .
So we need to look up and we need to look left of Square F.
Therefore as you can see we are going to be repeating a TON OF WORK, repeating the same recursive calls for all squares.

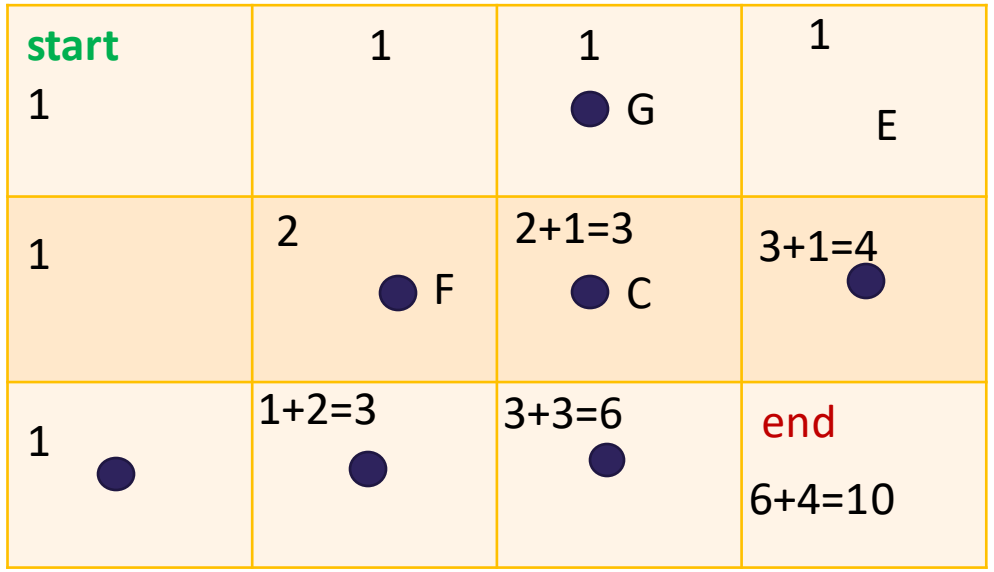
IN CONCLUSION: With this recursive approach we are not storing the results of the intermediate calculations and that in the long run is going to create a real bad run time complexity

Our recursive approach vs what we just talked about

We can only get to these squares on the border with 1 way. There is exactly one way to get these square

width =4

height =3



So this approach that we will carry recursively is upper bounded by
$O(2^{(n+m)})$ where **n is the width**, **m is the height.**
This is not the exact time complexity; it will be very close.

It is a poor algorithm. Bad solution, will take long among time..
The reason we have this run time complexity is :

Why Base 2? We look at 2 directions. (up and left)
Why n+m? to reach the base case we need to make at most n+m calls to get to a base case.
For example from the end mark we make a recursive call to the left, then left again, then up…It does not matter which way, it will not take more than n+m calls to reach to the base case.

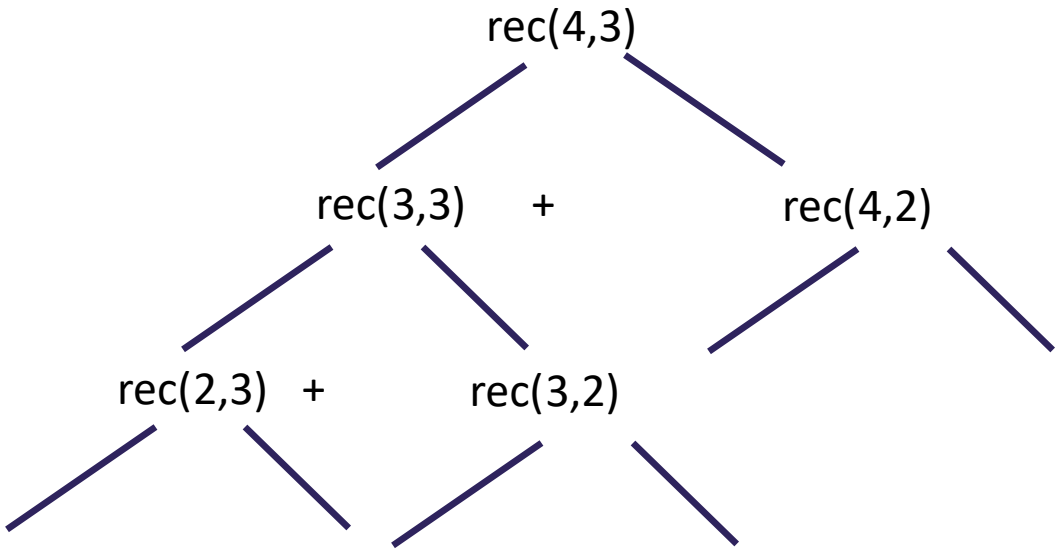
Complete this recursive function

```
def number_of_ways_to_get_to_end(width, height):
    pass
```

Recursive sol

```python
# Recursive solution
# O(2^(n+m)) run time
# O(n+m) space
def number_of_ways_to_get_to_end(width, height):
    if width == 1 or height == 1:
        return 1
    return number_of_ways_to_get_to_end(width - 1, height) + number_of_ways_to_get_to_end(width, height - 1)
```

Let us rename our algo number_of_ways_to_get_to_end it as rec so we can show the recursive tree



Run time = $O(2^{(n+m)})$

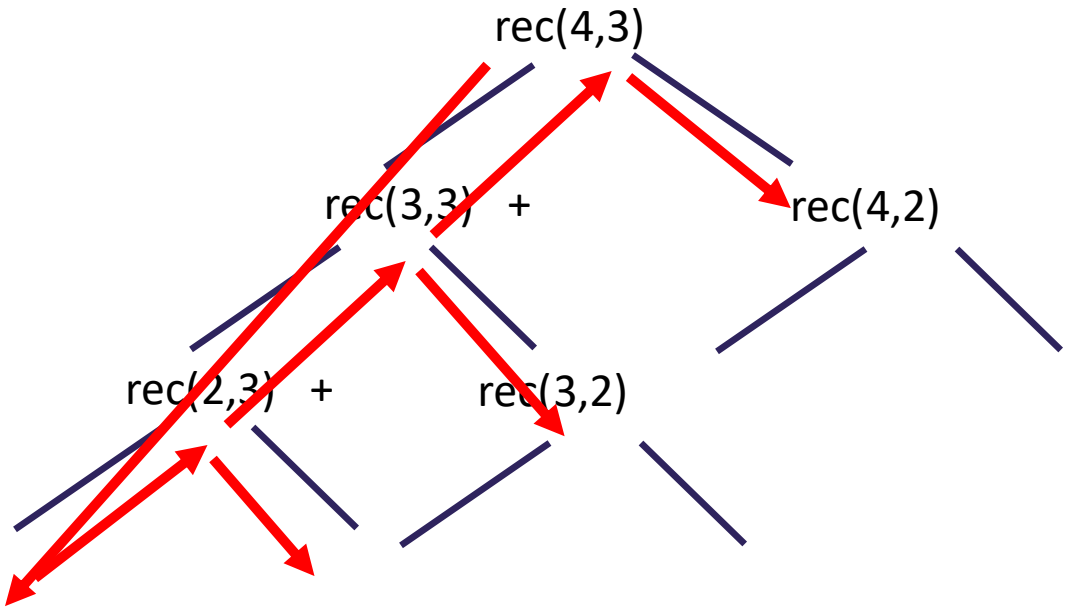
We will not branch out more than n+m times.
Height of our tree is no more than n+m
Because that is how long it will take us to reach to base case.
The reason we have 2^ is because we are branching out only twice, 2 directions only , one to check left, one to check up.

Space complexity is O(n+m) because we will reach to base case after (n+m) calls, or at most n+m calls.

Recursive sol

```python
# Recursive solution
# O(2^(n+m)) run time
# O(n+m) space
def number_of_ways_to_get_to_end(width, height):
    if width == 1 or height == 1:
        return 1
    return number_of_ways_to_get_to_end(width - 1, height) + number_of_ways_to_get_to_end(width, height - 1)
```

Let us rename our algo number_of_ways_to_get_to_end it as rec so we can show the recursive tree



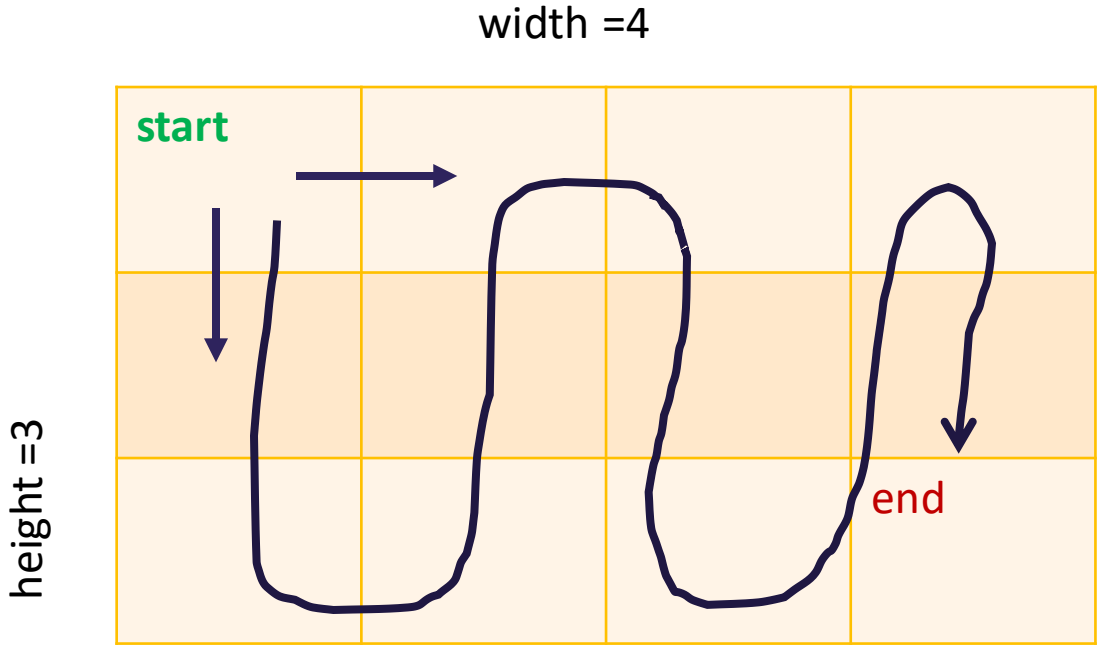
Space complexity is O(n+m) because we will reach to base case after (n+m) calls, or at most n+m calls.
We will go down one side, say we are going left, all the way down until it reaches its base case and then we slowly climb back up
So it goes as Depth first traversal until it reaches its base case. Then climbs back up slowly to all of the other sides.

We are never going to have more than the height of the tree recursive calls, in our recursive call stack.
So the call stack has no more than n+m calls.

Let’s make it better!

Dynamic Programming

Remember: You can either go down or right

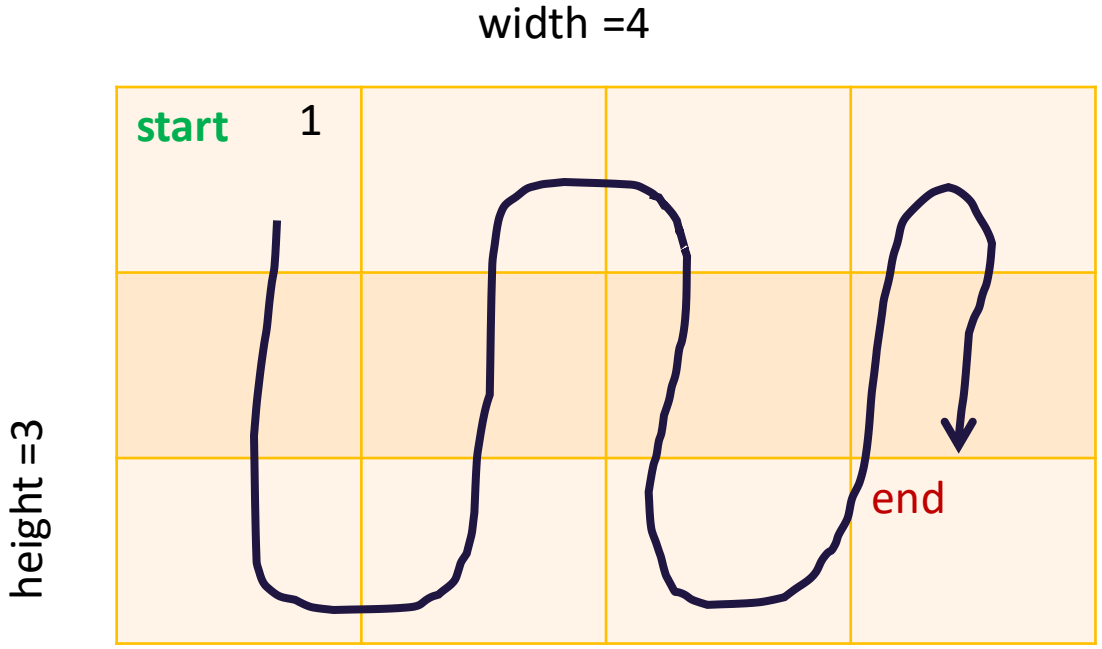


In previous solution we worked our way back up, started from End , proceed toward the Start,

With this solution we will approach from the Start point.

Dynamic Programming

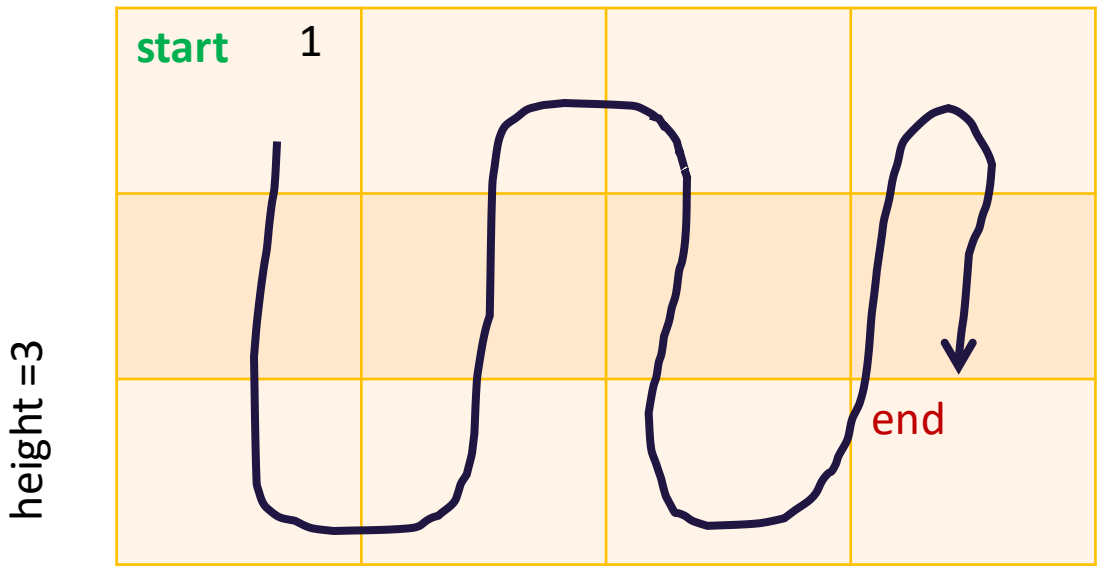
Remember: You can either go down or right



It takes only one way to get to starting position, we will loop downward, and update each value as we go

Dynamic Programming

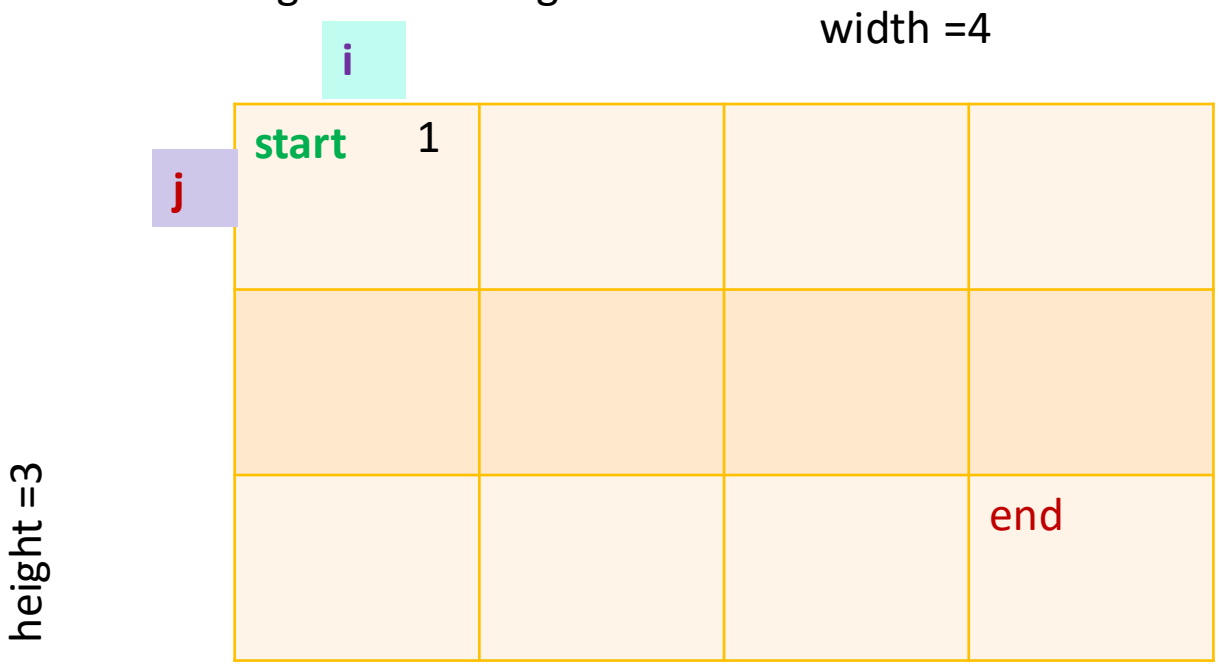
Remember: You can either go down or right



It takes only one way to get to starting position, we will loop downward, and update each value as we go. First thing we will do is to fill in our row and fill in our column, top border and the left border.

Dynamic Programming

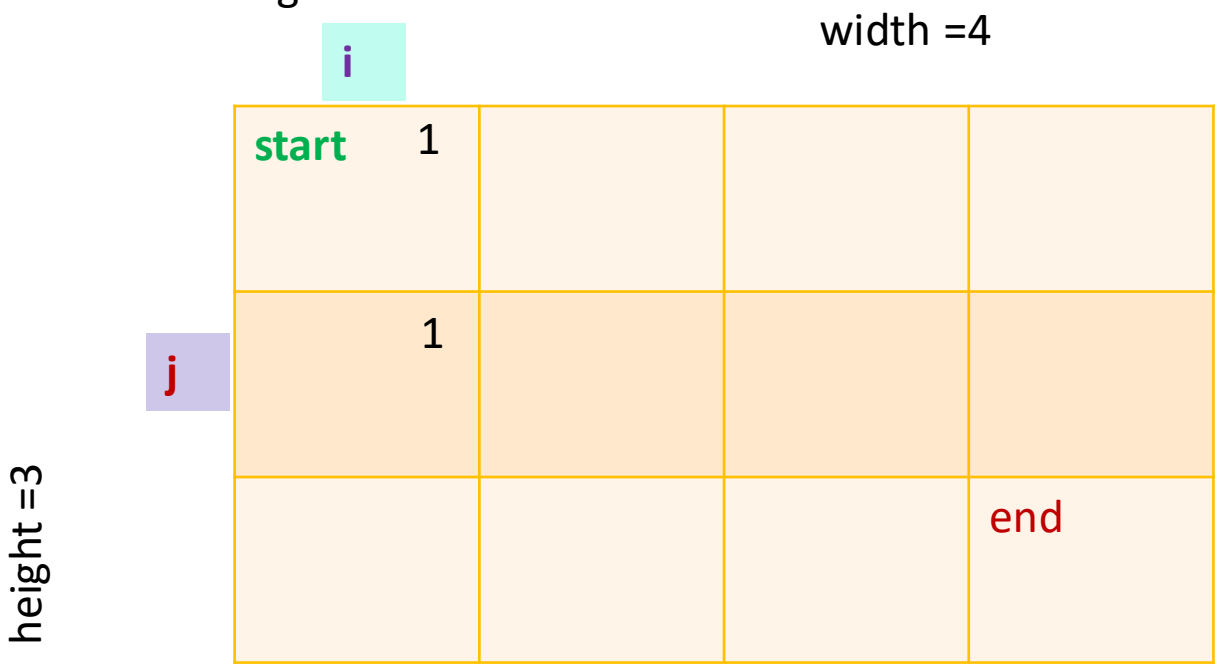
Remember: You can either go down or right



We are going to have 2 counters: **i and j**,
We will start with i=0 and j=0, update that cell so it is equal to 1, then we will move downward, increment j, so j becomes 1..

Dynamic Programming

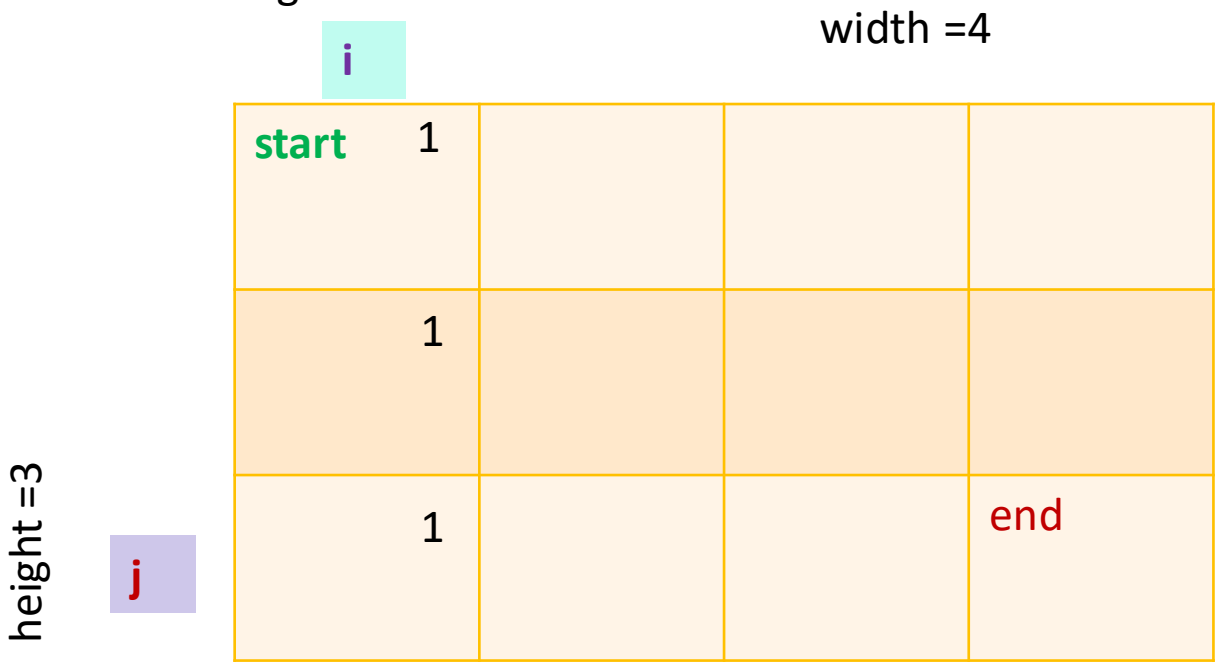
Remember: You can either go down or right



We are going to store how many ways we could get to this square, so what we are going to use is a 2-dimensional array to represent this grid and store information for each square.
We are on the border so the value is 1 for this square.

Dynamic Programming

Remember: You can either go down or right



We just finished the first column, now we will move i over, i=1..j = 0 again

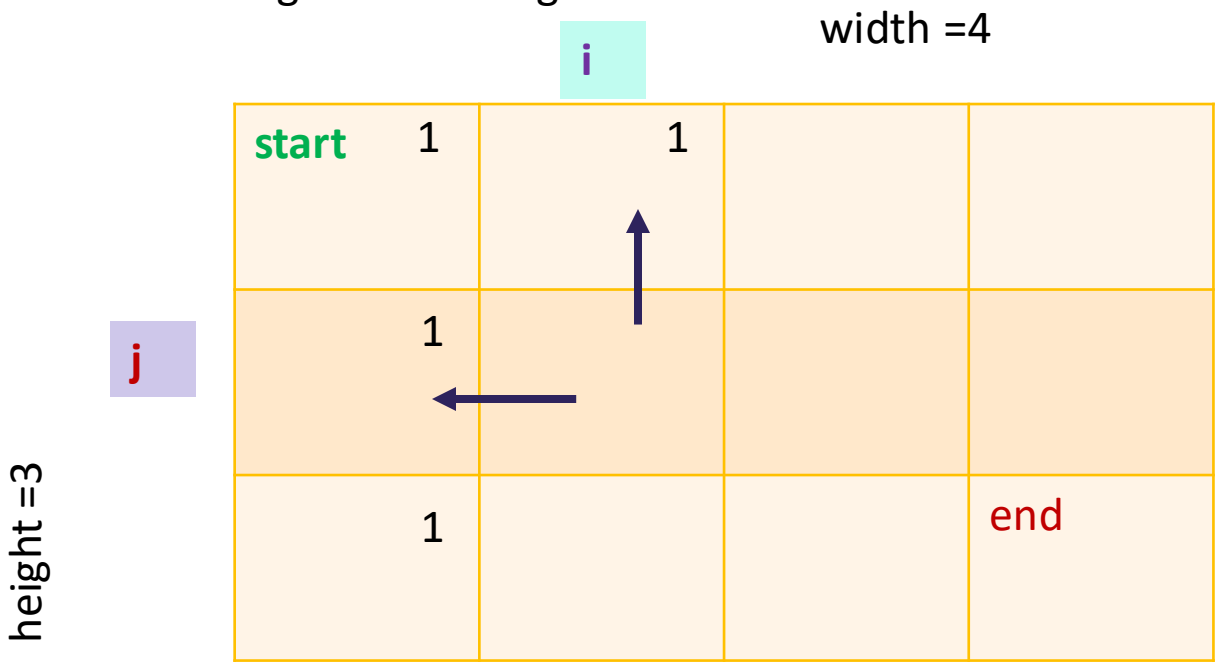
Dynamic Programming

Remember: You can either go down or right

We just finished the first column, now we will move i over.., j=0 , i=1 in this square the value is one, since it is on the border, now increment j

Dynamic Programming

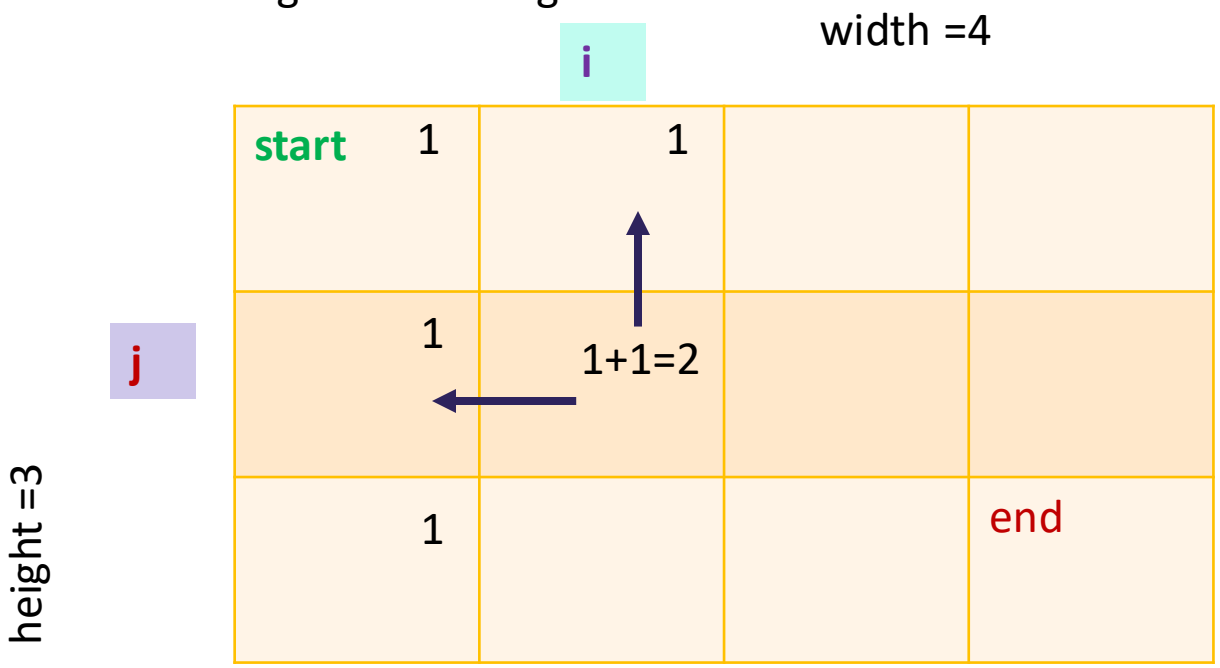
Remember: You can either go down or right



We now need to calculate this square, it is not a border case. All we need to do is look into the data structure on the left and look into data structure upward, add those 2 values to compute the number of ways to get to this square ☺

Dynamic Programming

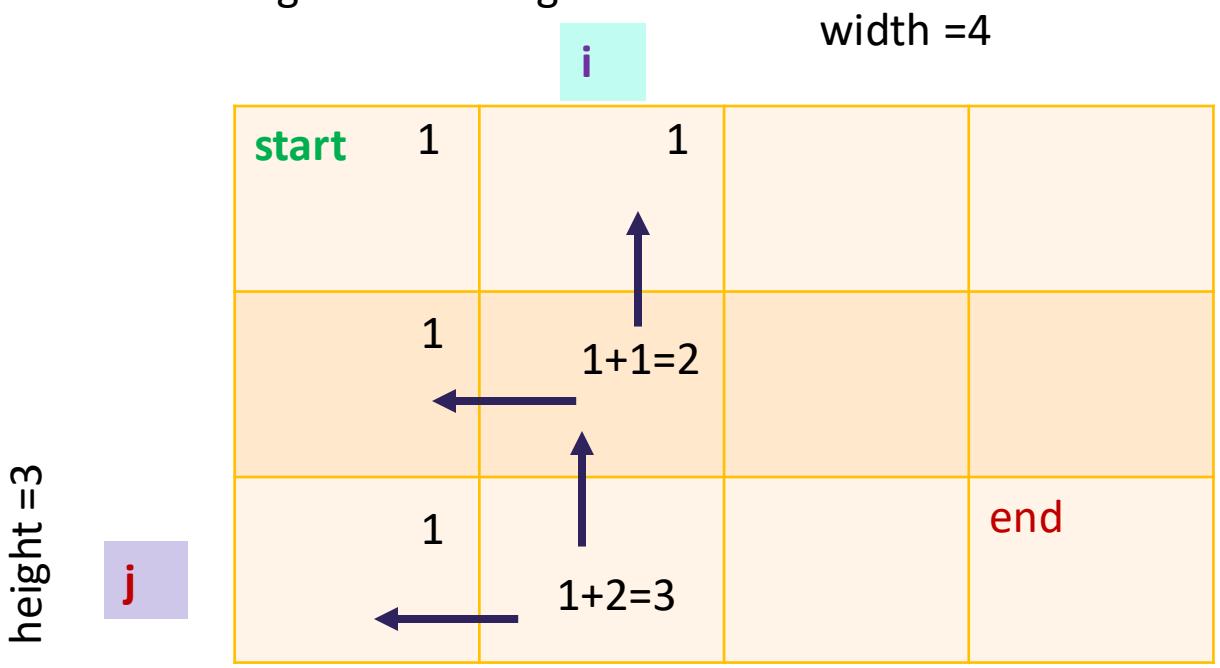
Remember: You can either go down or right



We now need to calculate this square, it is not a border case.
All we need to do is look into the data structure on the left and look into data structure upward, add those 2 values to compute the number of ways to get to this square ☺

Dynamic Programming

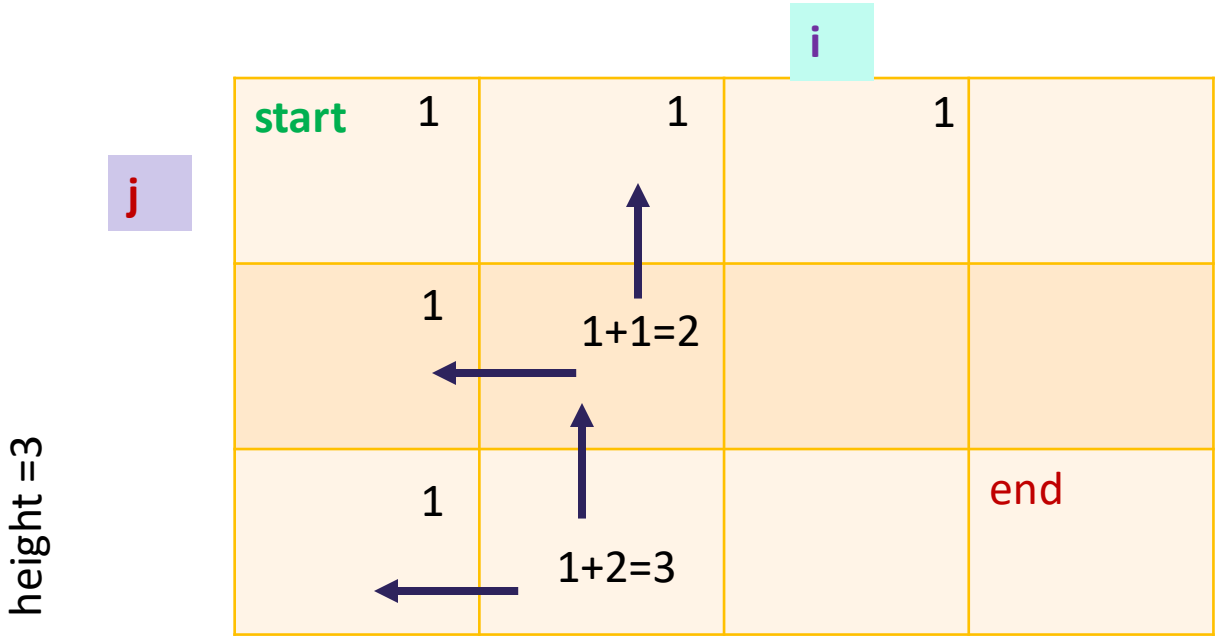
Remember: You can either go down or right



Dynamic Programming

Remember: You can either go down or right

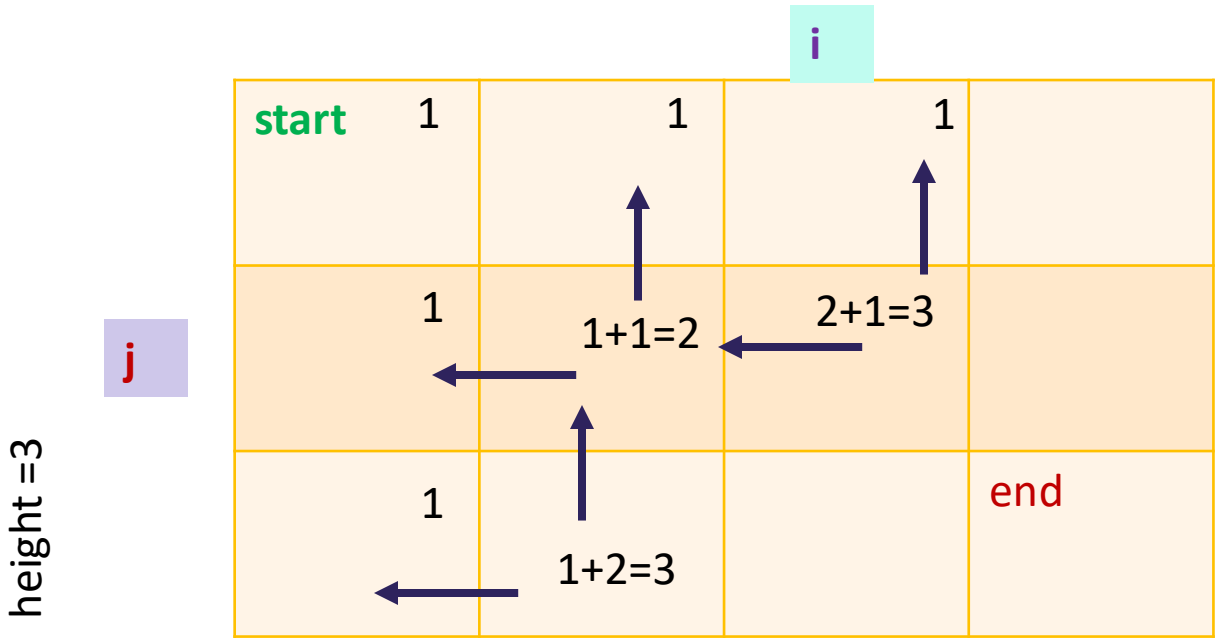
width =4



Dynamic Programming

Remember: You can either go down or right

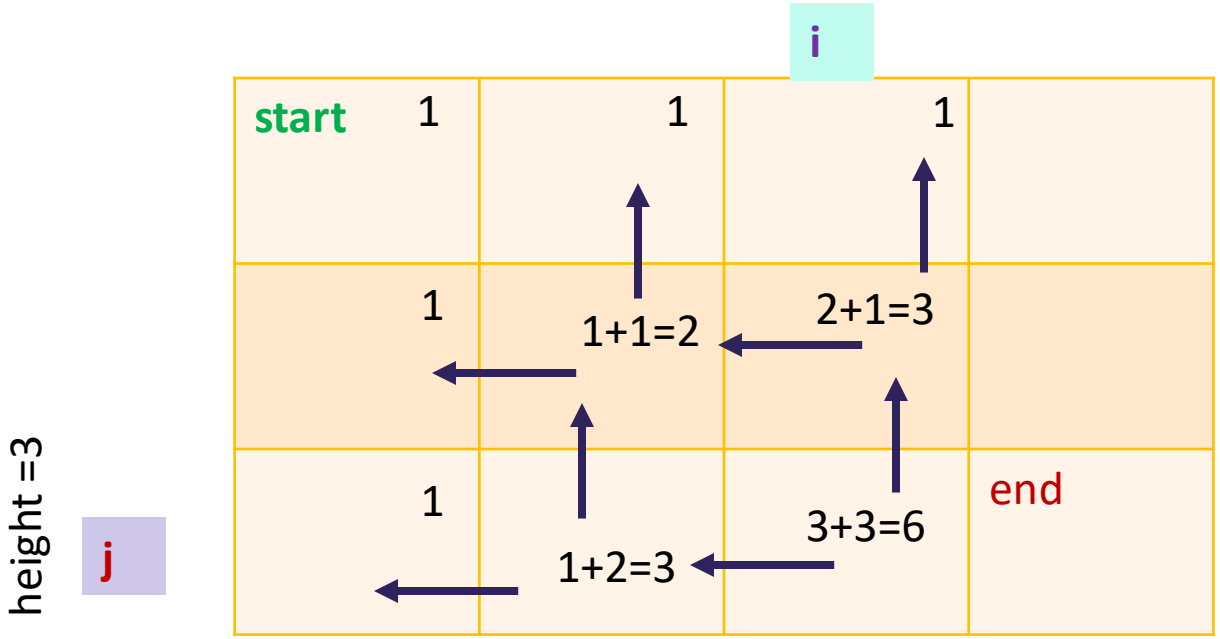
width =4



Dynamic Programming

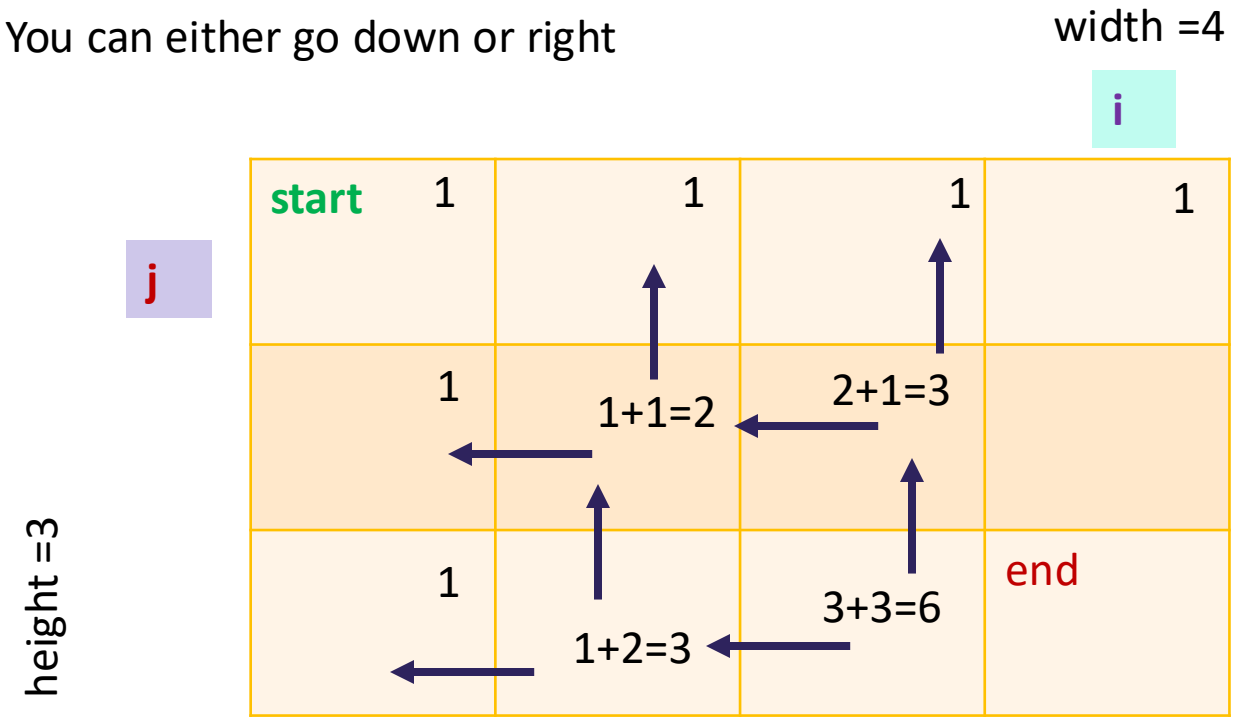
Remember: You can either go down or right

width =4



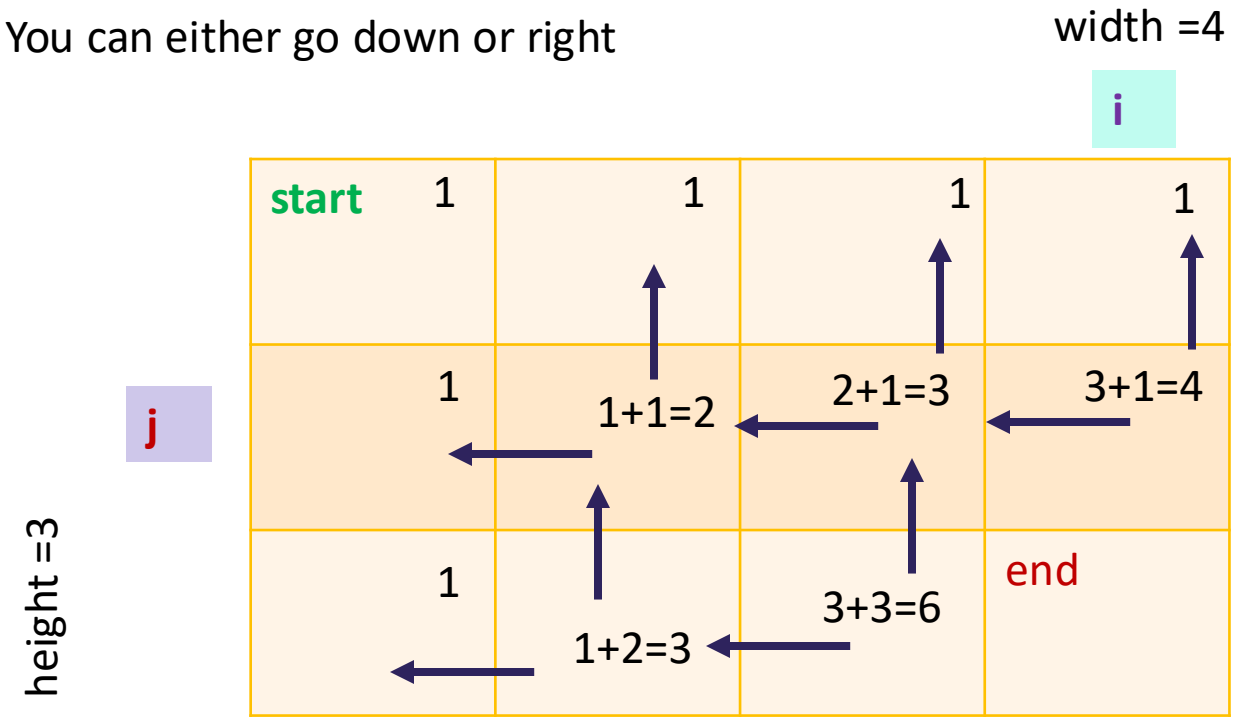
Dynamic Programming

Remember: You can either go down or right



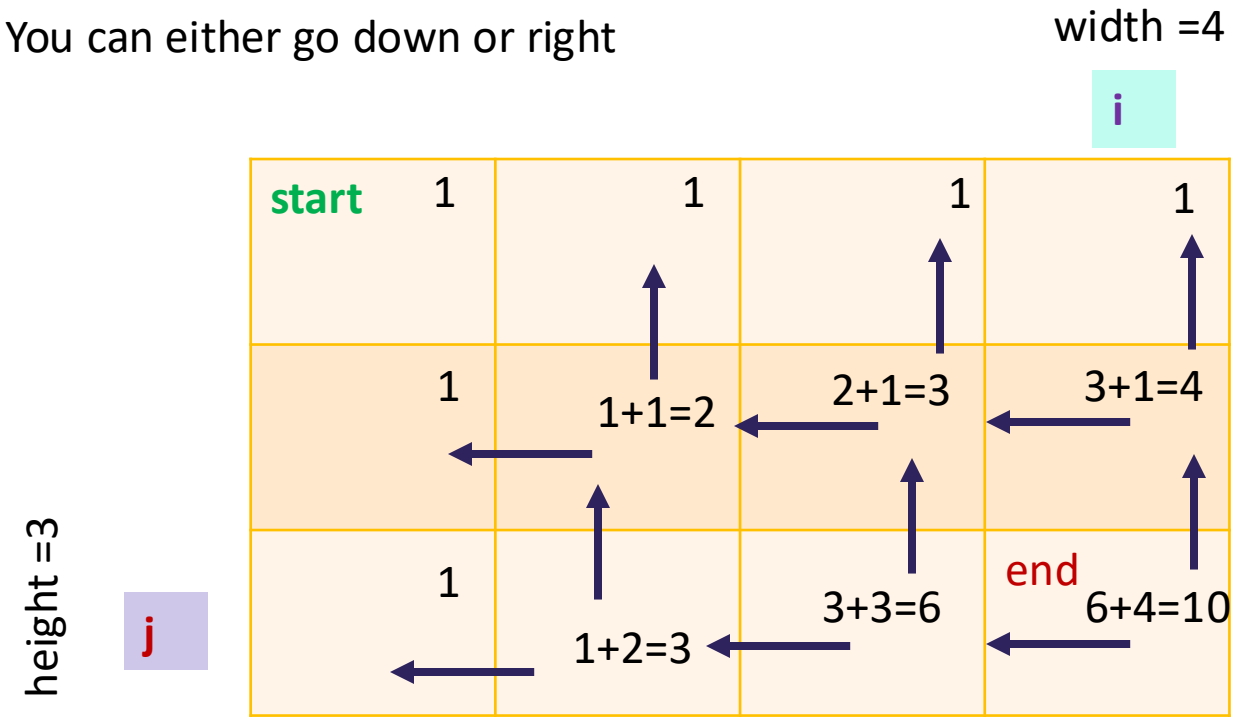
Dynamic Programming

Remember: You can either go down or right



Dynamic Programming

Remember: You can either go down or right



How to code this?

```
def number_of_ways_to_get_to_end(width, height):
    pass
    # Create a 2D list to keep track of the number of ways at each position


    # Loop through each position in the grid and calculate the number of ways

            # If the position is on the top row or left column, there is only one way to reach it

                # If the position is not on the top row or left column, calculate the number
                # of ways to get to the current position by adding the number of ways to
                # the position to the left and the number of ways to the position above.
                ways_left = number_of_ways[height_index][width_index - 1]  # looking to the left
                ways_up = number_of_ways[height_index - 1][width_index]  # looking up
                number_of_ways[height_index][width_index] = ways_left + ways_up

    # Return the number of ways to get to the bottom right corner of the grid
```

O(n*m) Run time
Reason n*m operations all the rows and all of the columns, we will loop n*m times
O(n*m) space Space because we are making a grid.

There is a better way to solve this problem but requires a mathematical formula and it is not very easy to think about it during an interview:
PERMUTATIONS, unique ordering in a set

Permutations

- A Set : {1,3,4}
- How many different ways can you represent this set:
- {1,4,3}
- {3,1,4}
- {4,1,3}

Number of ordering in this = number of unique ways you can possibly to create

Permutations

width =4

height =3

start 1	1	1	1
1	2	3	4
1	3	6	end 10

We have 2 movements, Right and Down:
If our width =4 how many times can we move until we can get to end?
If our height =3 same question applies here…

From the start Right movements =3 until the end of the grid
Down movements =2 until the end of the grid.
So, we have {Right, Right, Right, Down, Down}
Combinations of these can be calculated with a formula: $\frac{(Right + Down)!}{Right! * Down\ !}$

This solution runs in O (n+m) time and it will take O(1) space.