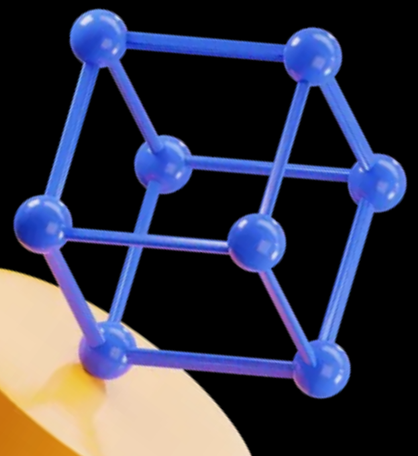
Hacker Devils

Sohum Sheth, Dean Richardson, Ahmed Shaheen

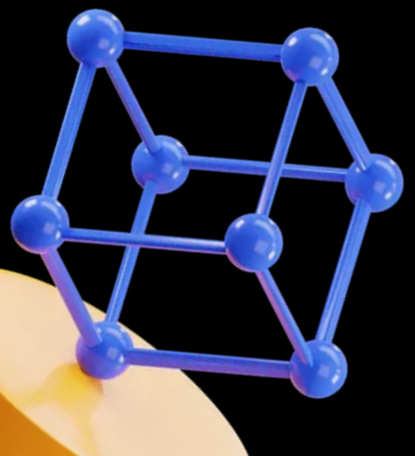
HKD MEDIUM SECTION #5

Intro to Dynamic Programming

March 23rd, 2026



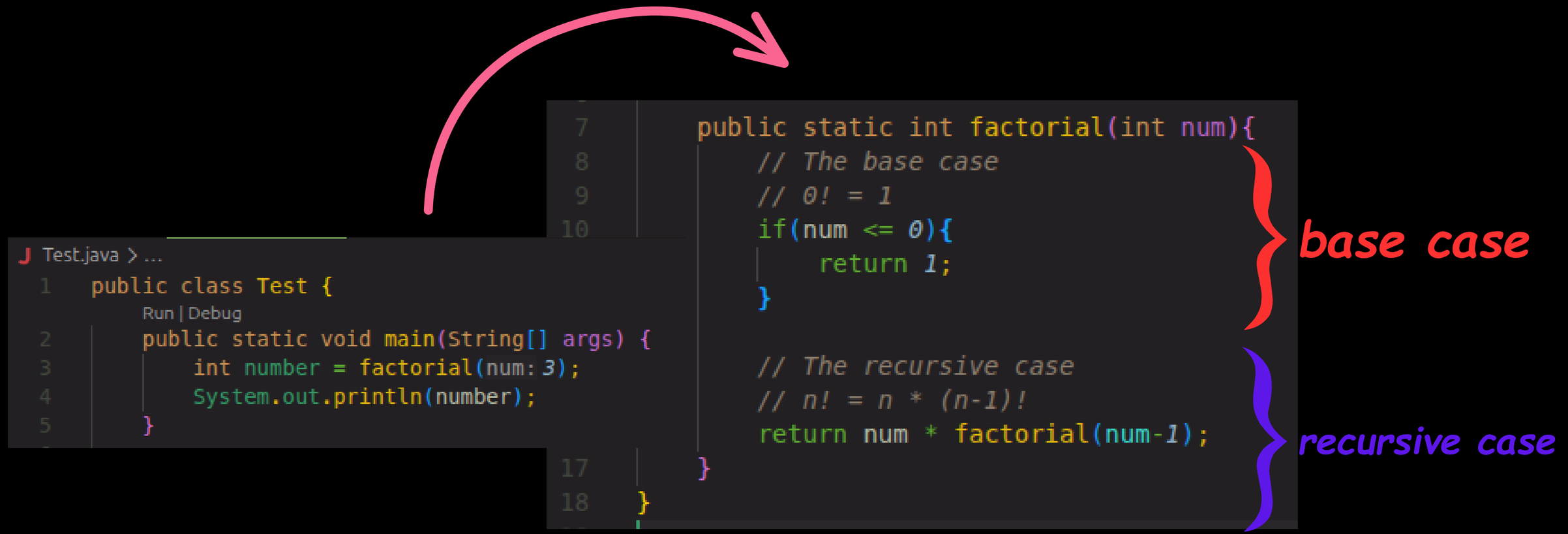
Check-In!



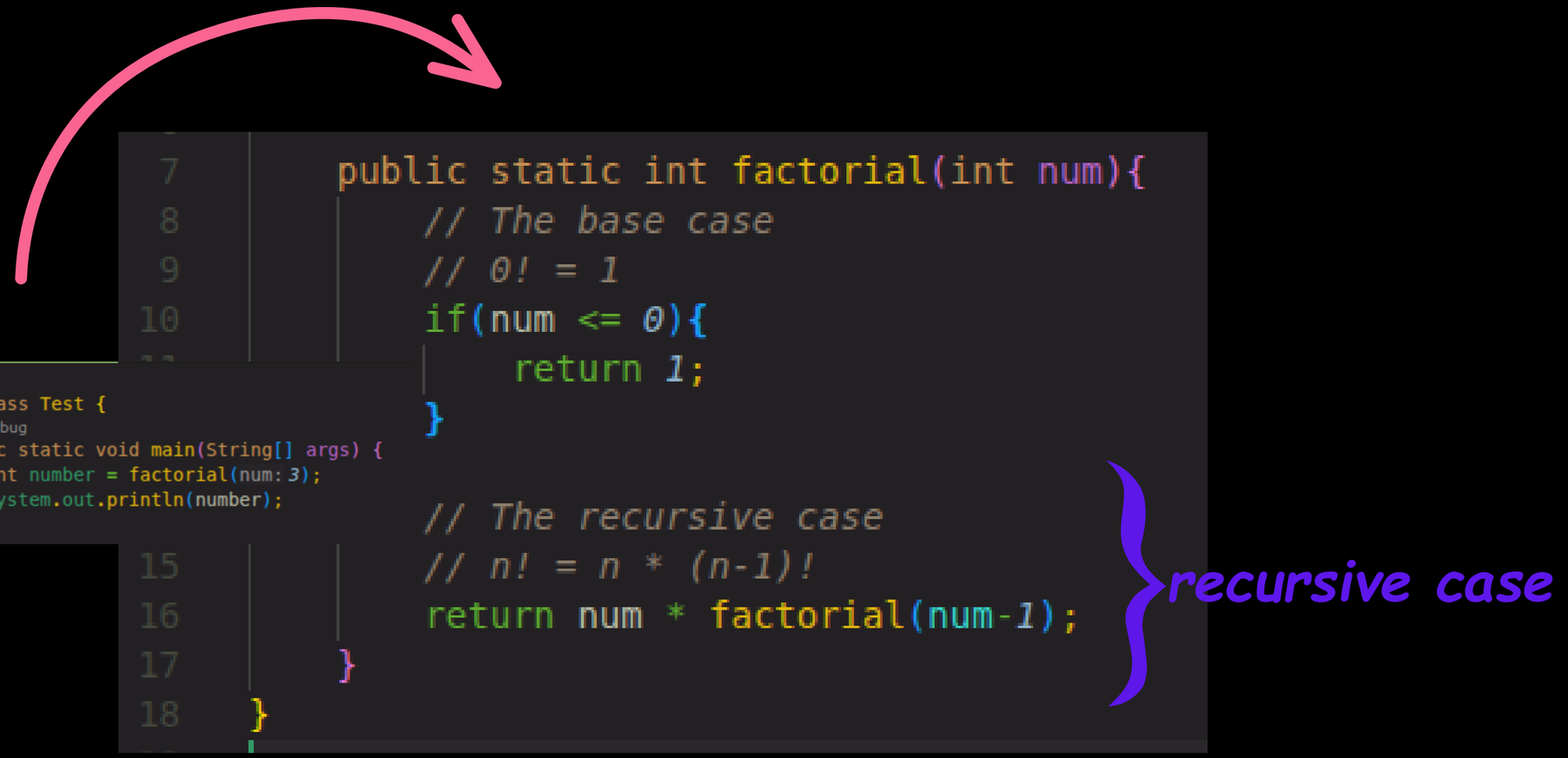
Before we get into Dynamic Programming...
We have to first learn Recursion!

But what is Recursion,
and how exactly do I implement it?

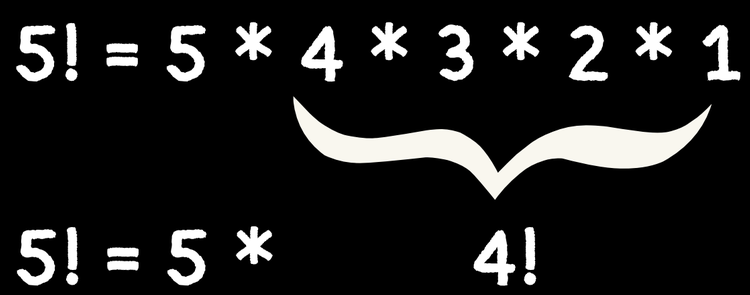
Recursion is defined as
a function that calls itself repeatedly
until some “escape” condition is met.



But what does that mean??
I like to first start with the recursive case.

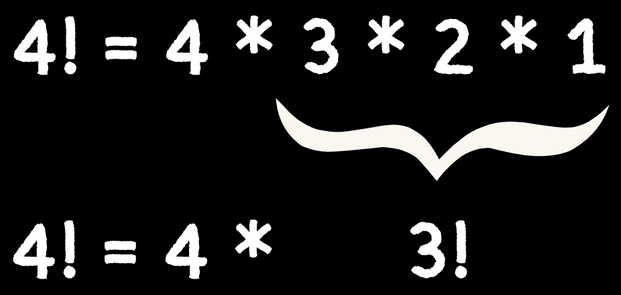


Since we are implementing solving a factorial using recursion, I want to **write out the function first**.



If you didn't notice, 5! = 5 * 4!

Same goes for 4!... 4! = 4 * 3!

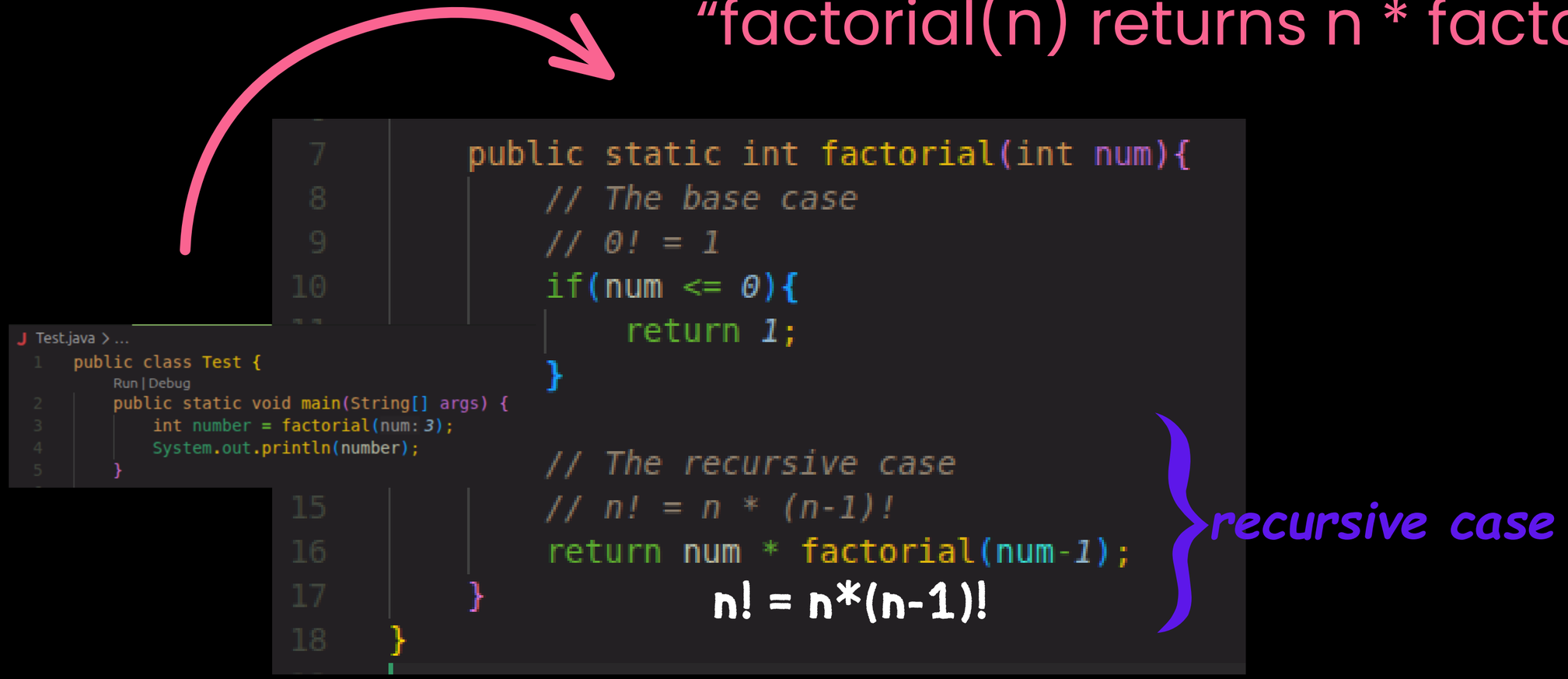


This means that...

$$n! = n*(n-1)!$$

Well, awesome! we have our recursive case!

n! = n*(n-1)!
directly translates to
“factorial(n) returns n * factorial(n-1)”



Alright! We got our recursive case...
now let's get our base case.

The base case is really important;
otherwise your code may end up
looping forever!

To find our base case, let's think of
another property of factorials:

```
1! = 1 * 0! = 1
0! = 1
```

This means that...

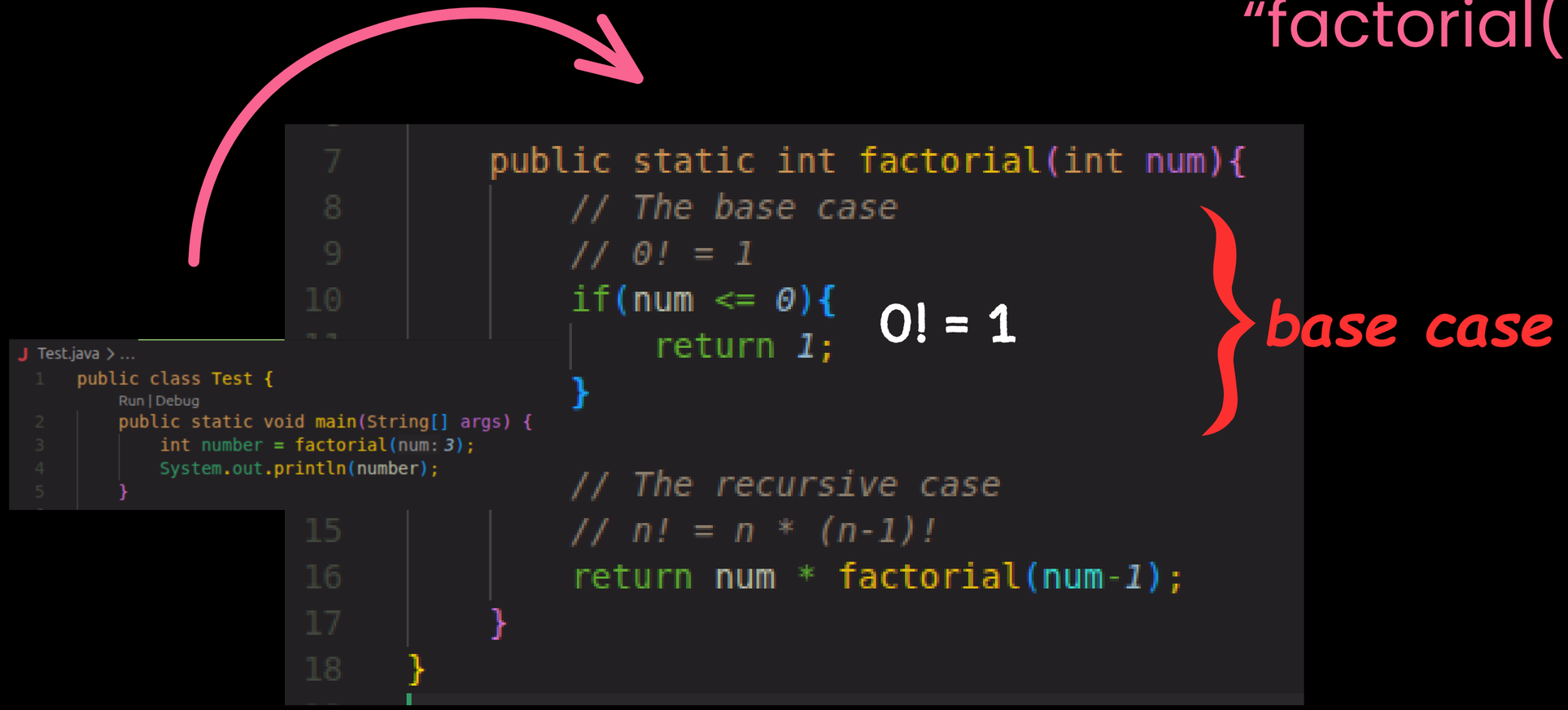
```
0! = 1
```

Let’s implement the base case.

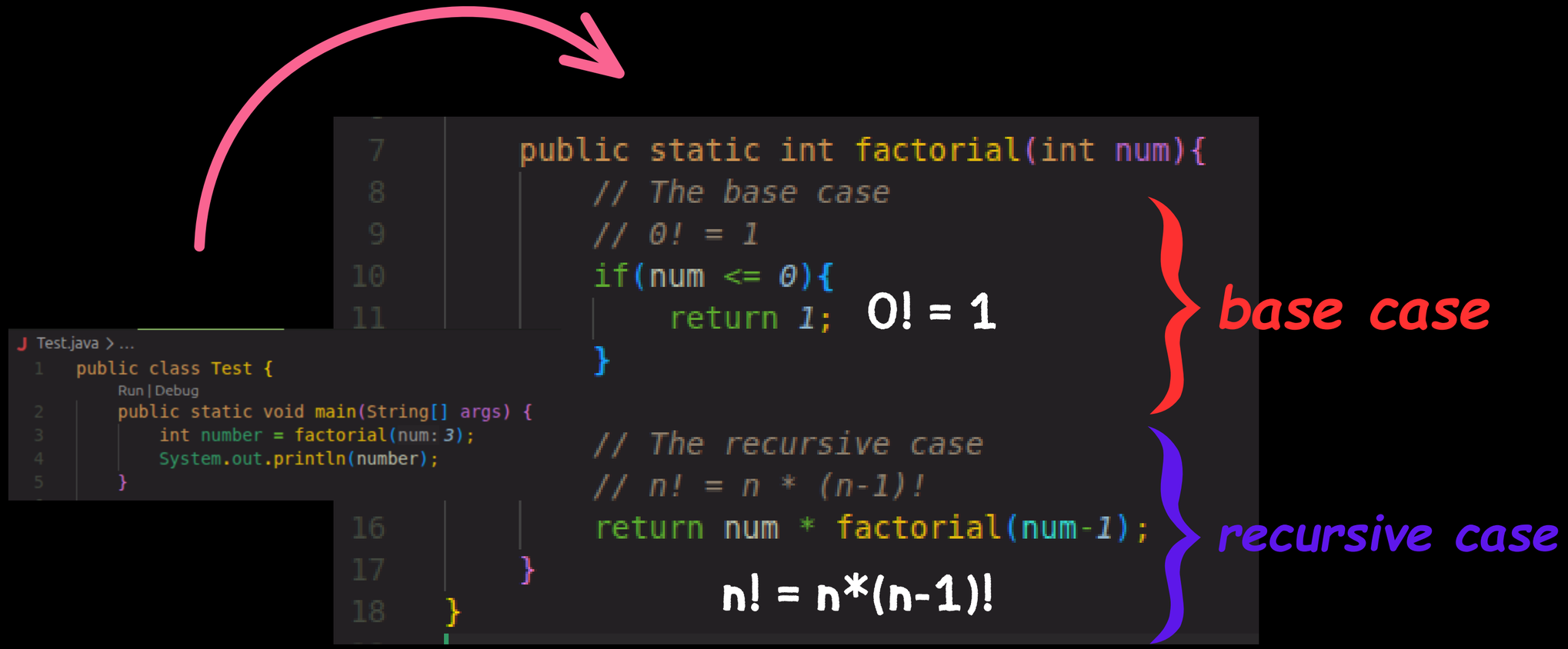
0! = 1

directly translates to

“factorial(0) returns 1”



And we're done!



Some things to note

Test.java > ...

```java
public class Test {
    Run | Debug
    public static void main(String[] args) {
        int number = factorial(num: 3);
        System.out.println(number);
    }

    public static int factorial(int num){
        // The base case
        // 0! = 1
        if(num <= 0){
            return 1;
        }

        // The recursive case
        // n! = n * (n-1)!
        return num * factorial(num-1);
    }
}
```

You may notice how this is just one function, so how do I know what is being called by main and what is being called by the function?

Answer: you don't! The main method is **simply starting at a point in the recursive process**.

(all the function knows is to return a value to **who** called it.)



f(6) → f(5) ↔ f(4) ↔ f(3) ↔ f(2) ↔ f(1) ↔ f(0)

return f(5) return f(4) return f(3) return f(2) return f(1) return f(0) return x;

70. Climbing Stairs

Easy | Topics | Companies | Hint

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

- `1 <= n <= 45`

The Warmup

https://leetcode.com/problems/climbing-stairs/

remember that you will need both the { *base case* *recursive case*

Note: your code *will* time out!
this is intentional!!!

Warmup Solution

In this sort of problem, it sometimes helps to find the base case first/**start small and work your way up**...

`n = 1`



You are climbing a staircase. It takes `n` steps to reach the top.

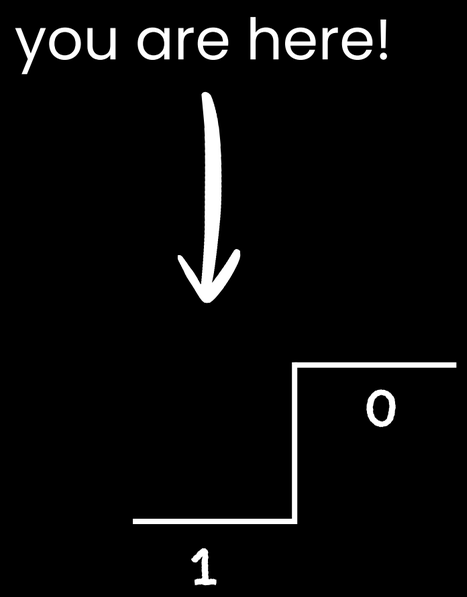
Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Warmup Solution

In this sort of problem, it sometimes helps to find the base case first/**start small and work your way up**...

`n = 1`

Of course, you will start at step = 0. According to the problem, how many ways can you ascend?



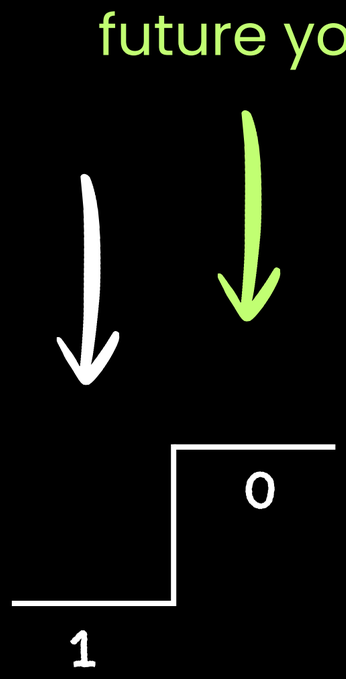
You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Warmup Solution

In this sort of problem, it sometimes helps to find the base case first/**start small and work your way up**...

`n = 1`



Of course, you will start at step = 0. According to the problem, how many ways can you ascend?

You can only ascend once, right? That means we return 1 (for one possibility)

You are climbing a staircase. It takes `n` steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

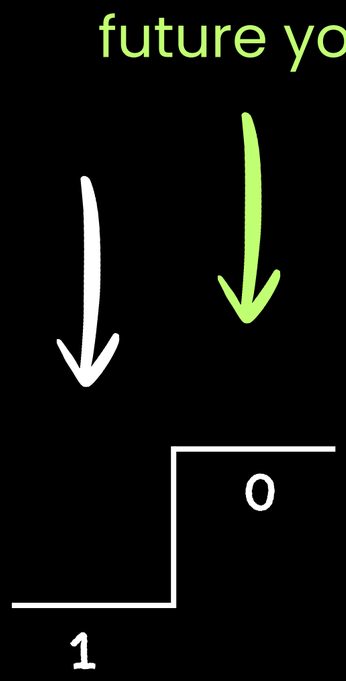
Warmup Solution

In this sort of problem, it sometimes helps to find the base case first/**start small and work your way up**...

Base cases:

n=1 → returns 1

`n = 1`



This seems like a base case to me.
Let's write it down!

You are climbing a staircase. It takes `n` steps to reach the top.

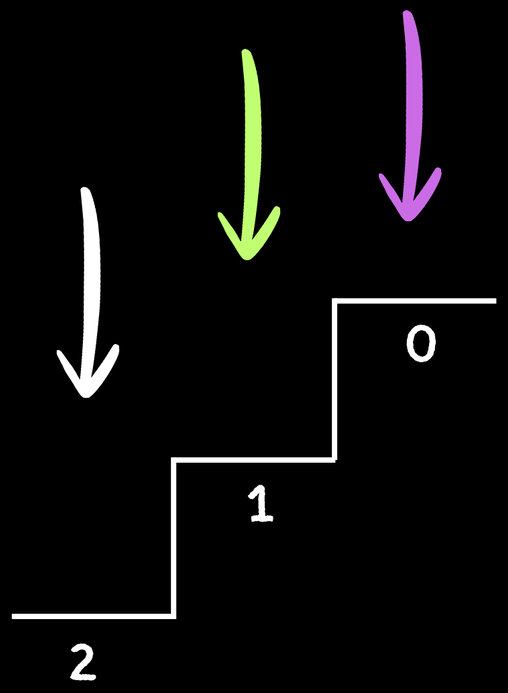
Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Warmup Solution

Base cases:

n=1 → returns 1

n = 2



Let's move on to n=2. It seems like I can either move up 1 or 2 steps.

If I move up **2 steps**, I'm at the top! That's 1

If I move up **1 step**, I'm at the second-last step, then I move up 1 again. That's another.

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

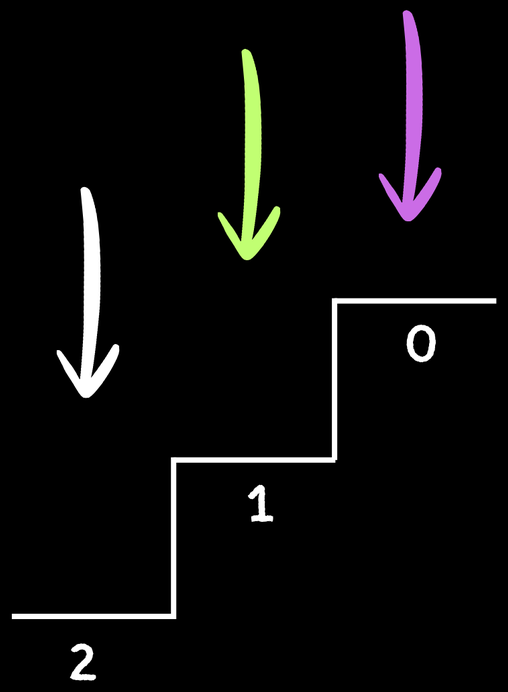
Warmup Solution

Base cases:

n=1 → returns 1

n=2 → returns 2

n = 2



In total, that's **2 possibilities!**

Since I have two possibilities when n = 2,
I can record my 2 possibilities
in my base cases!

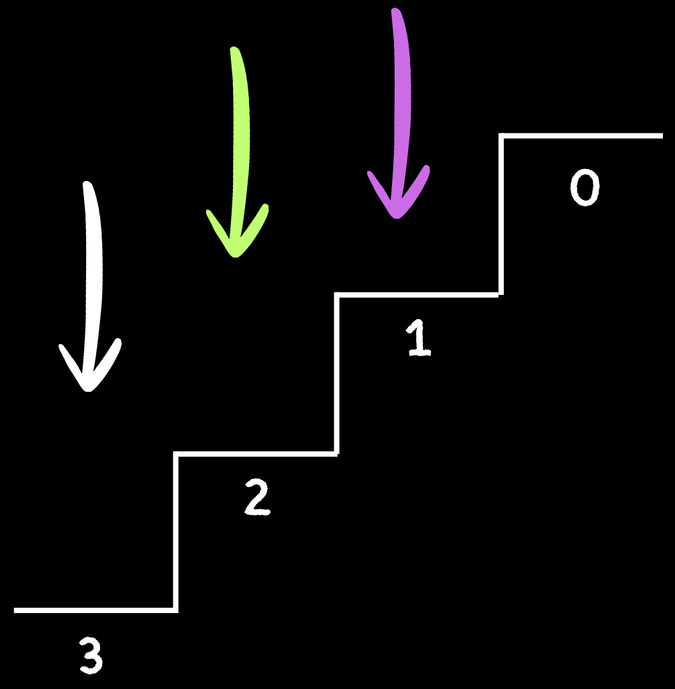
You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb `1` or `2` steps. In how many distinct ways can you climb to the top?

Warmup Solution

Base cases:
n=1 → returns 1
n=2 → returns 2

n = 3



At n=3, it seems like we have just another base case to record.
But look no further!

At this stage, we can only step up once or twice.

But if you've been paying attention, if you step up either one or two steps, that's just our base cases! We've been here before!

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

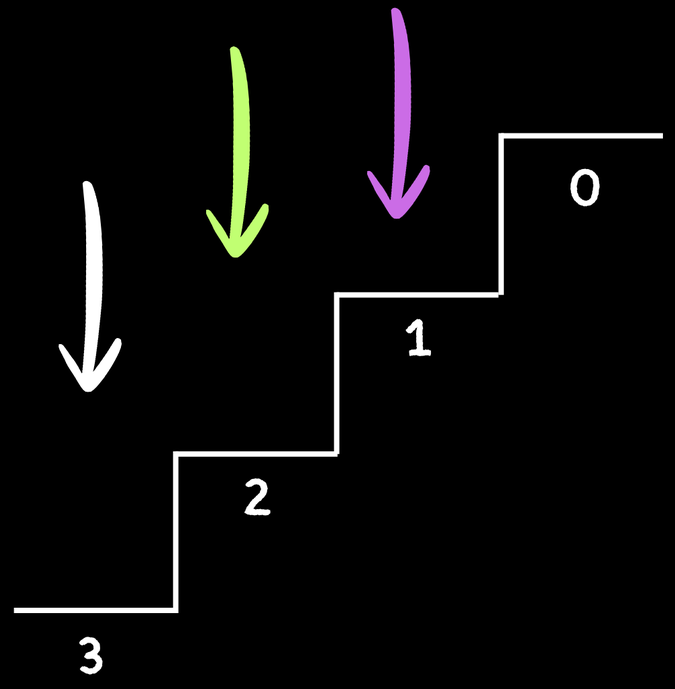
Warmup Solution

Base cases:

n=1 → returns 1

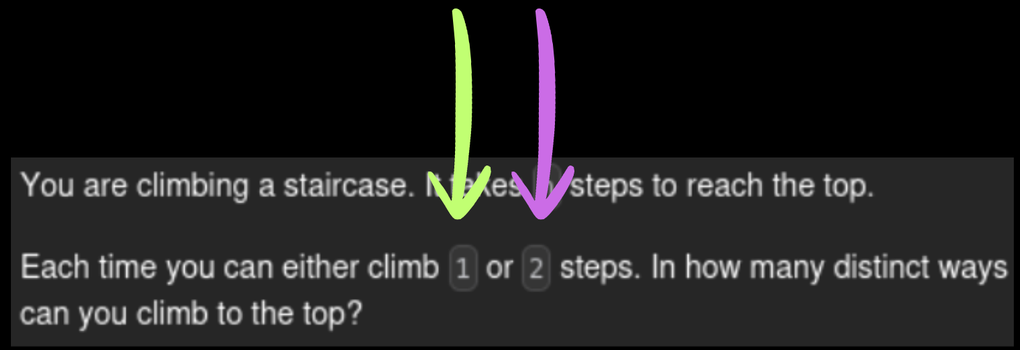
n=2 → returns 2

n = 3



Because we were here before, we can just do climbStairs(2) and climbStairs(1), referencing our base cases.

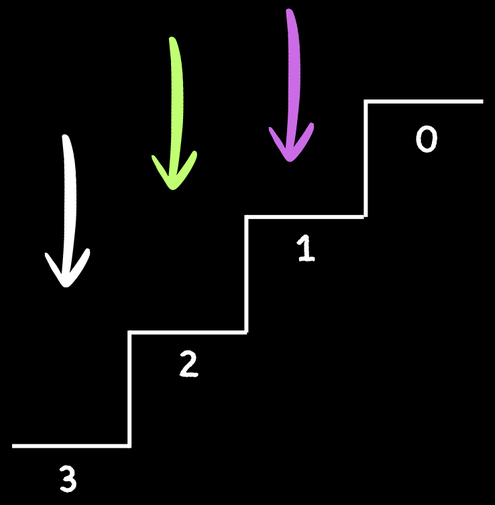
But taking a step back, that's just our recursive case, isn't it?!



Solution Formula

Recursive Case:

$$F(n) = F(n-1) + F(n-2)$$



Base Case:

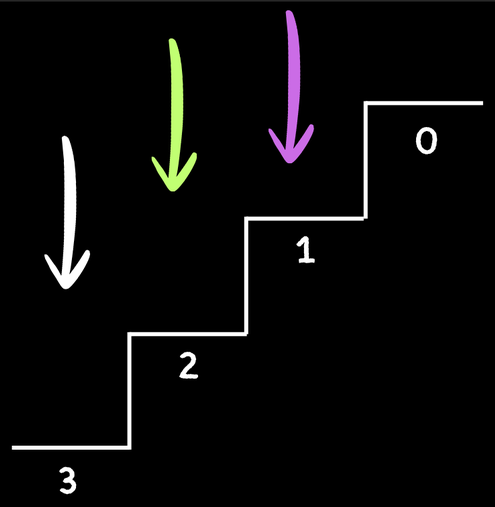
$$F(1) = 1$$

$$F(2) = 2$$

Warmup Solution (Java)

```java
class Solution {
    public int climbStairs(int n) {
        // The base case(s)
        if(n == 1){
            return 1; // 1 step
        }
        if(n == 2){
            return 2; // 1+1 step or 2 step
        }

        // The recursive case
        return climbStairs(n-1) + climbStairs(n-2);
    }
}
```



Recursive Case:
$F(n) = F(n-1) + F(n-2)$

Base Case:
$F(1) = 1$
$F(2) = 2$

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        # The base case(s)
        if n == 1:
            return 1  # 1 step
        if n == 2:
            return 2  # 1+1 step or 2 step

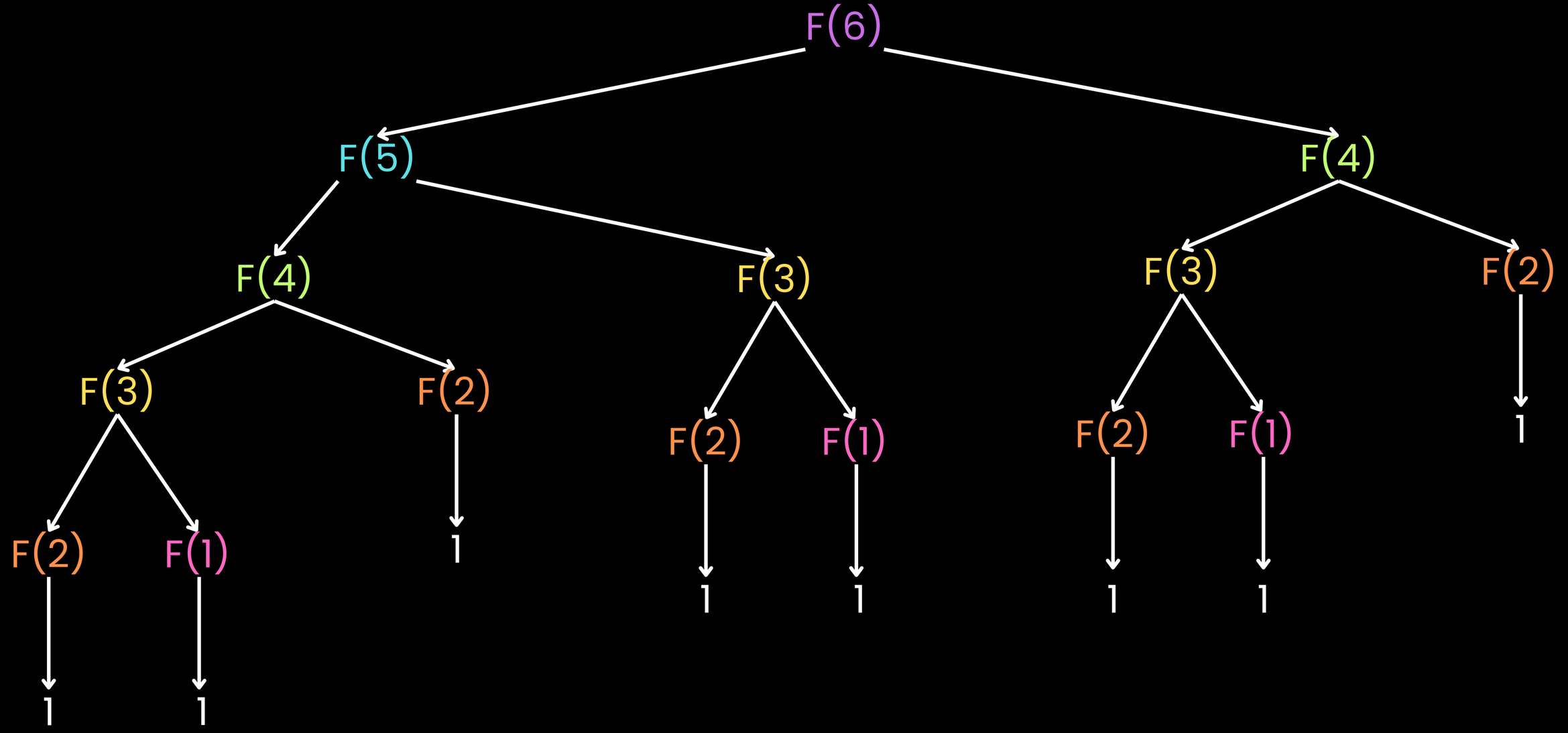
        # The recursive case
        return self.climbStairs(n - 1) + self.climbStairs(n - 2)

```

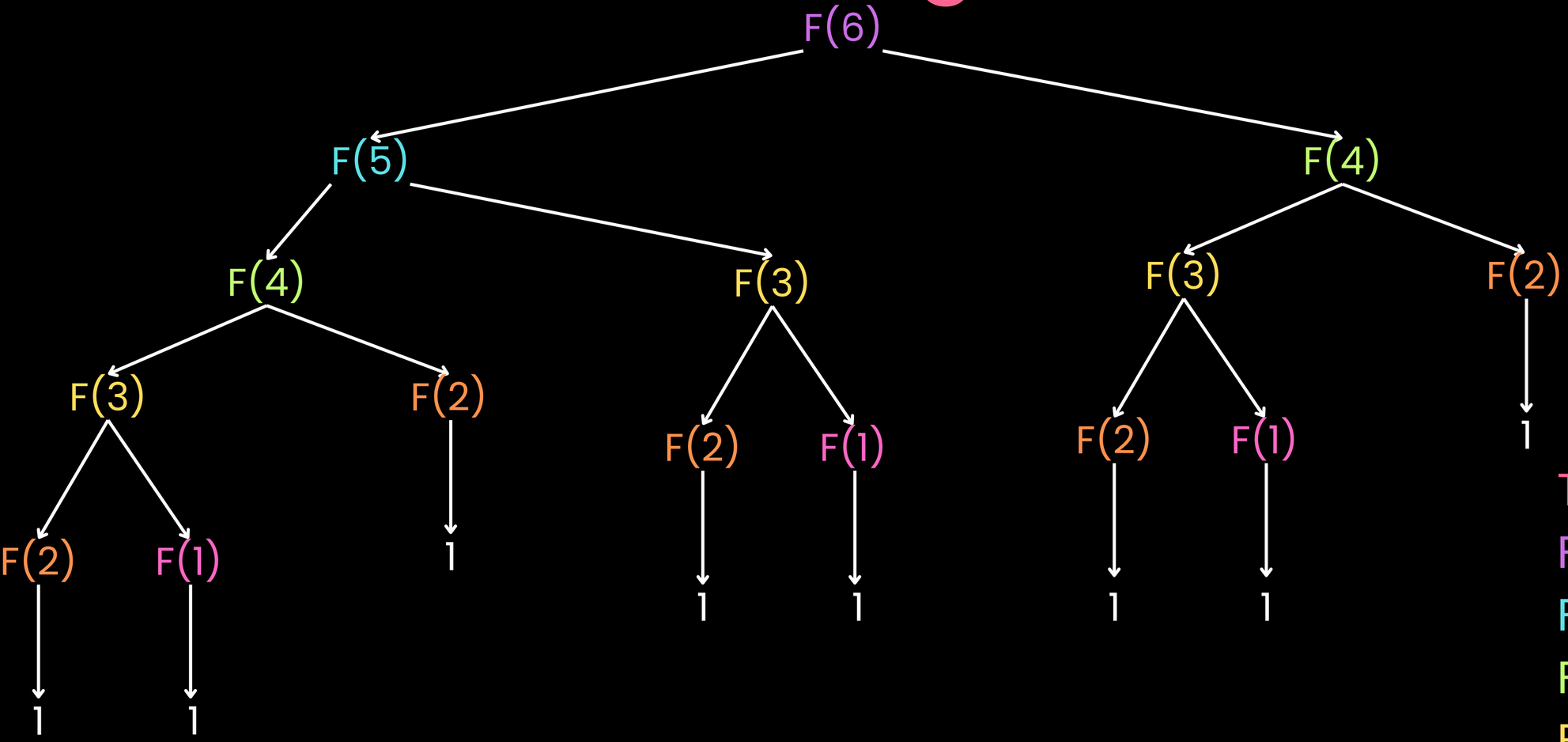
Warmup Solution (Python)

Illustrating the Solution

for n = 6



Illustrating the Solution



Total Function Calls:

F(6): 1

F(5): 1

F(4): 2

F(3): 3

F(2): 5

F(1): 3

Problem?

Solution is Highly Inefficient

Total Function Calls:
F(6): 1
F(5): 1
F(4): 2
F(3): 3
F(2): 5
F(1): 2

It can be shown to be $O(2^n)$

Meaning, that each function call takes about two times the number of operations as the one before

For example: F(7) is about two times the number of operations as F(6)

F(8) is two times the number of operations as F(7)

Problem?

THIS IS VERY BAD

Even Worse

Total Function Calls:
F(6): 1
F(5): 1
F(4): 2
F(3): 3
F(2): 5
F(1): 2

We are redoing so many operations!

We are calculating each function multiple times, when we only need to calculate it once

These are called **Overlapping Subproblems**

Solution:
Dynamic Programming

Dynamic Programming

Recursion

Total Function Calls:
F(6): 1
F(5): 1
F(4): 2
F(3): 3
F(2): 5
F(1): 2

Dynamic Programming

Total Function Calls:
F(6): 1
F(5): 1
F(4): 1
F(3): 1
F(2): 1
F(1): 1

Dynamic Programming

Recursion

Total Function Calls:
F(6): 1
F(5): 1
F(4): 2
F(3): 3
F(2): 5
F(1): 2

Dynamic Programming

Total Function Calls:
F(6): 1
F(5): 1
F(4): 1
F(3): 1
F(2): 1
F(1): 1

Dynamic Programming

Recursion

Total Function Calls:
F(6): 1
F(5): 1
F(4): 2
F(3): 3
F(2): 5
F(1): 2

Dynamic Programming

Total Function Calls:
F(6): 1
F(5): 1
F(4): 1
F(3): 1
F(2): 1
F(1): 1

How?

Dynamic Programming: An algorithmic technique that optimizes brute-force recursion by breaking a problem down into smaller subproblems, solving each subproblem just once, and storing their solutions.

To illustrate...

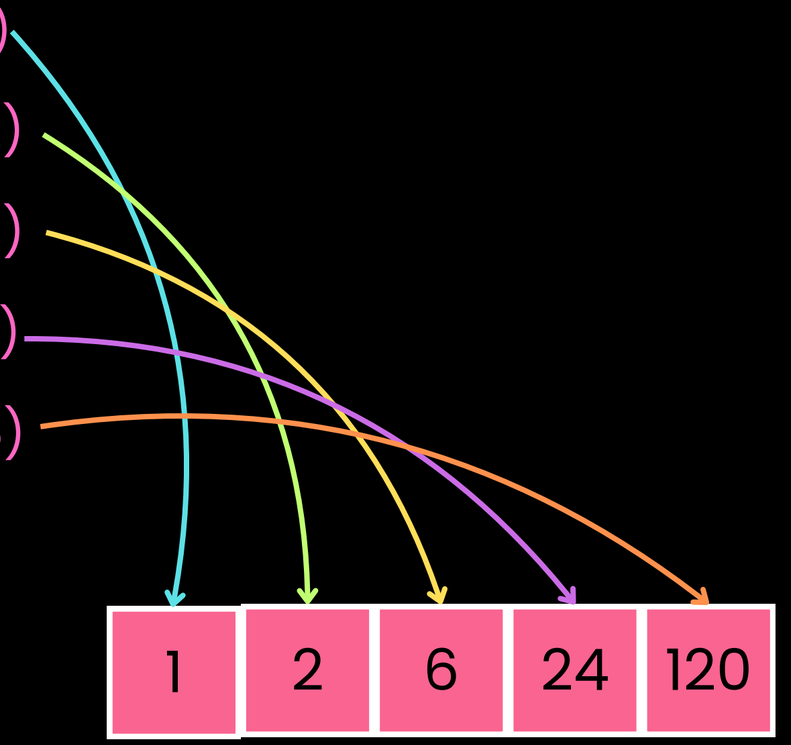
What's 5x5?

You didn't calculate 5+5+5+5+5, you just solved that problem long ago, stored the solution in your brain, and pulled it out right now!

More examples

Write a function that calculates the factorial: F(n) = n!

	Recursion	Dynamic Programming	
F(1):	= 1	= 1	Store F(1)
F(2):	= 1 * 2	= F(1) * 2	Store F(2)
F(3):	= 1 * 2 * 3	= F(2) * 3	Store F(3)
F(4):	= 1 * 2 * 3* 4	= F(3) * 4	Store F(4)
F(5):	= 1 * 2 * 3 * 4 * 5	= F(4) * 5	Store F(5)
F(6):	= 1 * 2 * 3 * 4 * 5 * 6	= F(5) * 6	
Operations:	16 Multiplications	5 Multiplications	

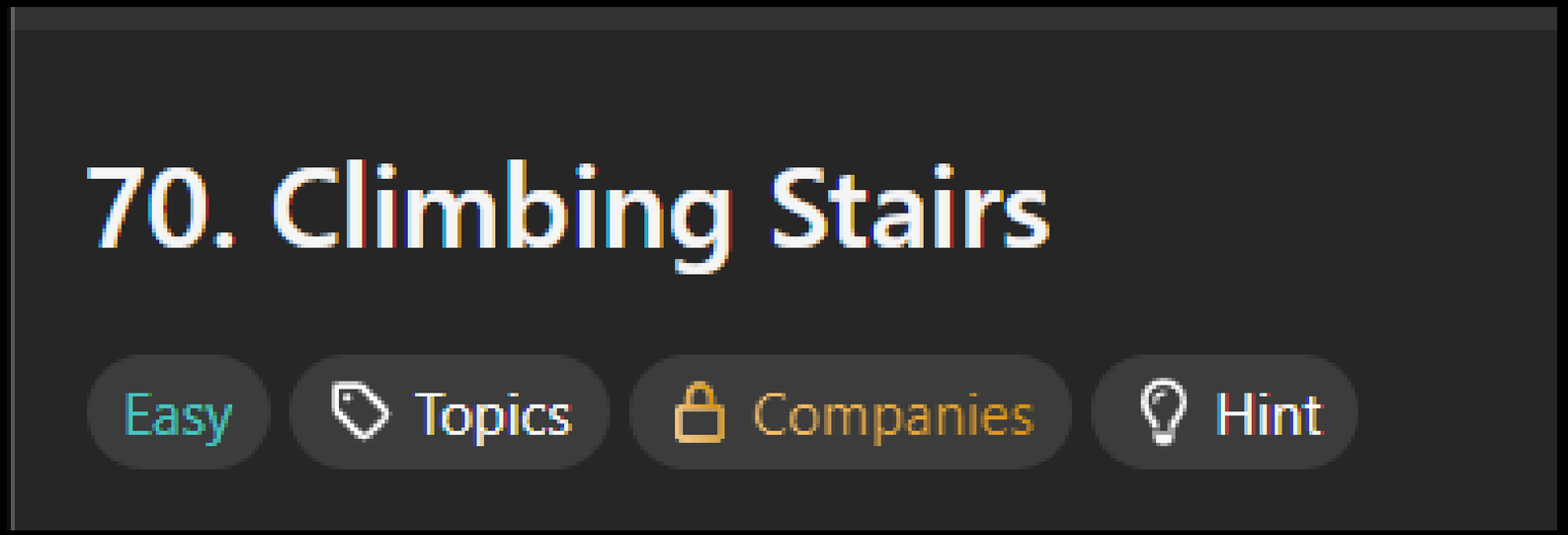


Definition

Bottom-Up DP (or Tabulation): The process of building up the solution to a large function (like the factorial) by starting at the bottom and iterating upwards, storing all of our solutions to avoid repeating calculations

One of the two-methods of DP, the other is a Top-Down approach (or Memoization) which is a little more complicated and wont be covered today

Challenge: Solve it Again



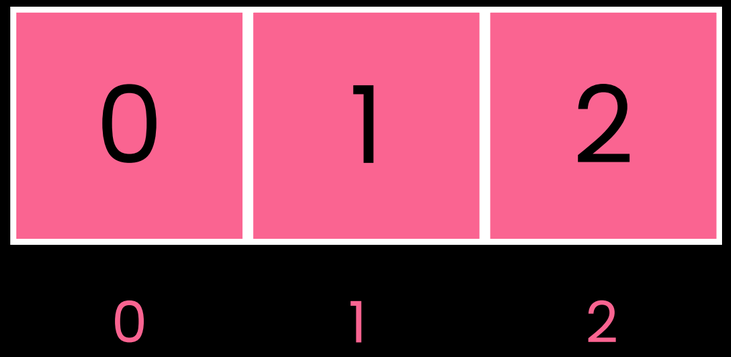
But... Use Dynamic Programming for a faster solution (Specifically, Bottom-Up DP)

Solution

If you remember from our previous solution, the base cases are:

F(1) = 1
F(2) = 2

Instead of recursing, we can add this to an array as follows



The zero at the start is just to keep indexes consistent with n values

Observation

Remember that for any given step (n), we can only get to that step from two steps:

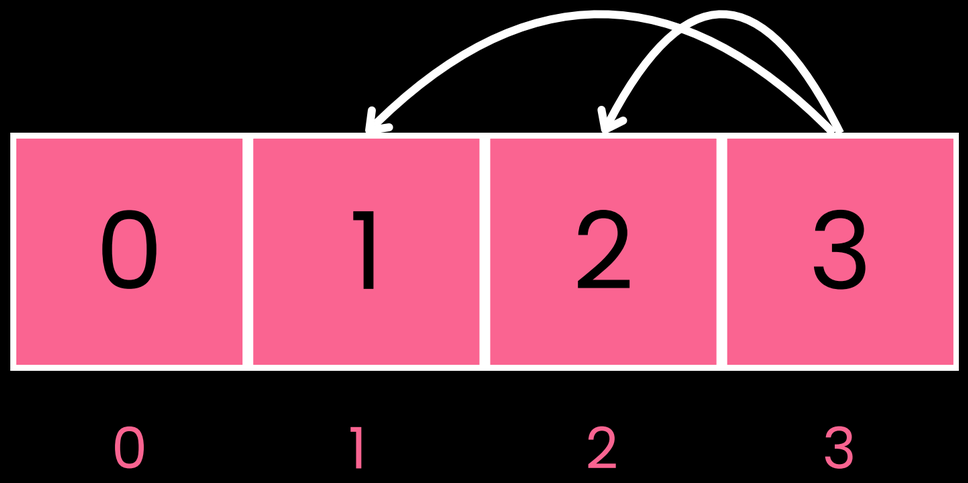
- The one right below it (n-1)
- The one two steps below it (n-2)

Then, the number of ways you can get to the step n is just the number of ways you can get to step n-1 added to that of step n-2

Filling in the array

Then: F(n) = F(n-1) + F(n-1)

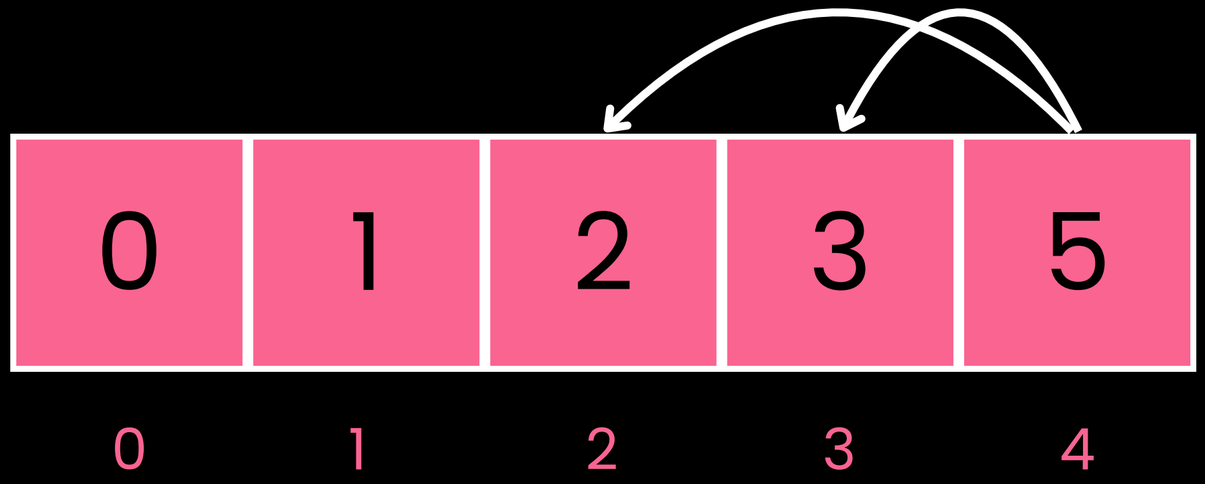
Remember, we got this same formula for recursion, but now instead of calling the function again, we can just use our stored values!



Filling in the array

Then: F(n) = F(n-1) + F(n-1)

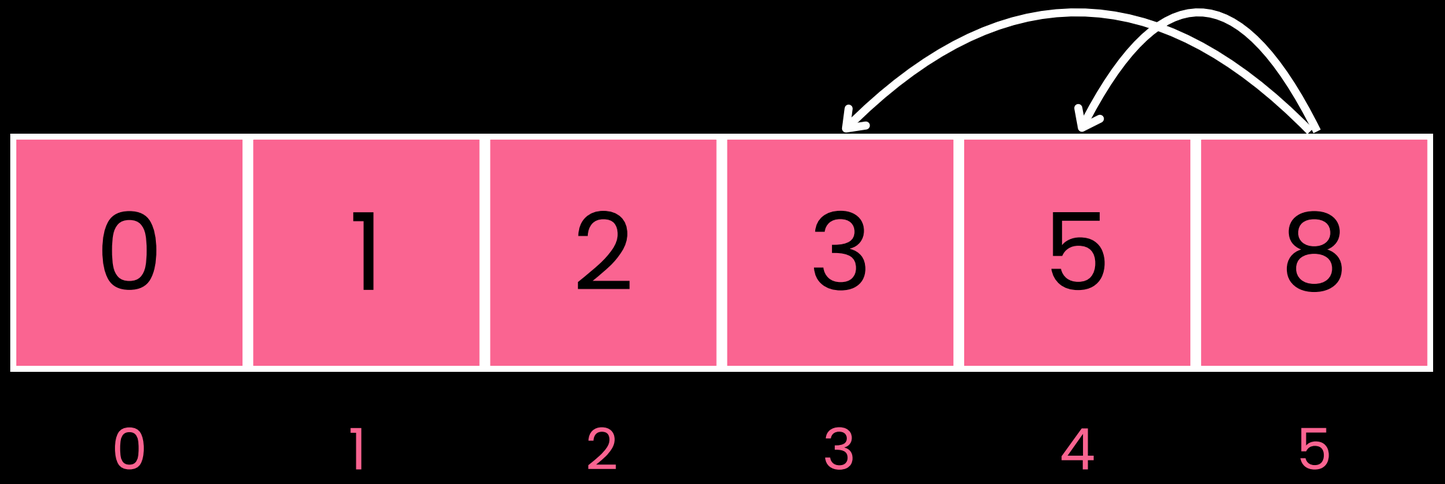
Remember, we got this same formula for recursion, but now instead of calling the function again, we can just use our stored values!



Filling in the array

Then: F(n) = F(n-1) + F(n-1)

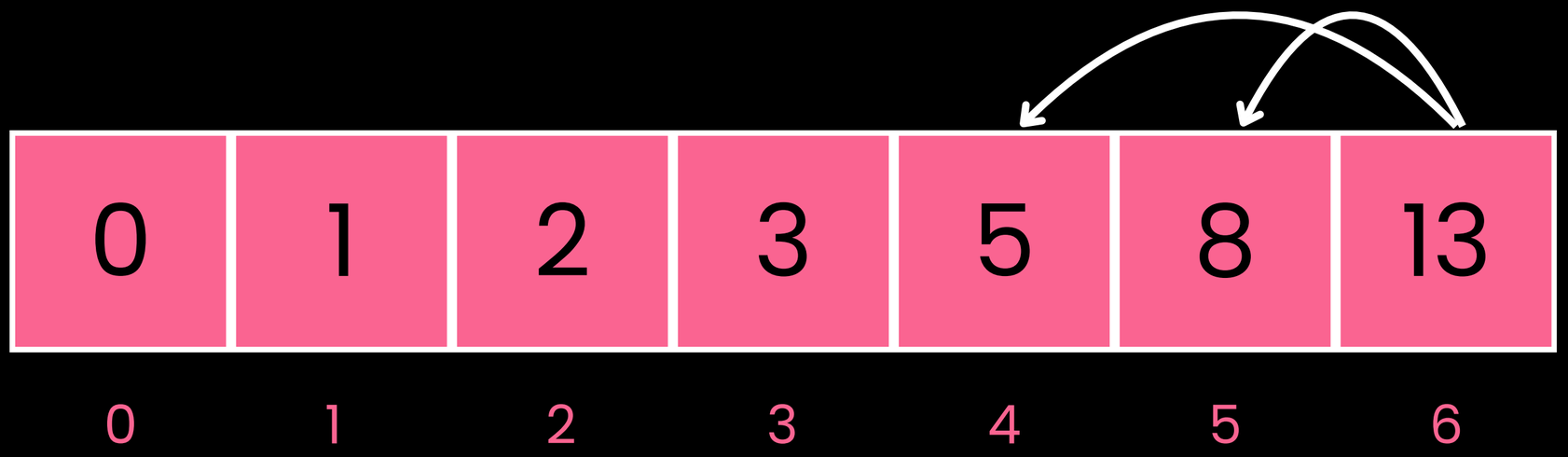
Remember, we got this same formula for recursion, but now instead of calling the function again, we can just use our stored values!



Filling in the array

Then: F(n) = F(n-1) + F(n-1)

Remember, we got this same formula for recursion, but now instead of calling the function again, we can just use our stored values!



Continue until we get our value.

Python Solution

```python
def climbStairs(n):
    if n <= 2:
        return n

    dp = [0] * (n + 1)

    dp[1] = 1
    dp[2] = 2

    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

Java Solution

```java
class Solution {
    public int climbStairs(int n) {
        if (n <= 2) {
            return n;
        }

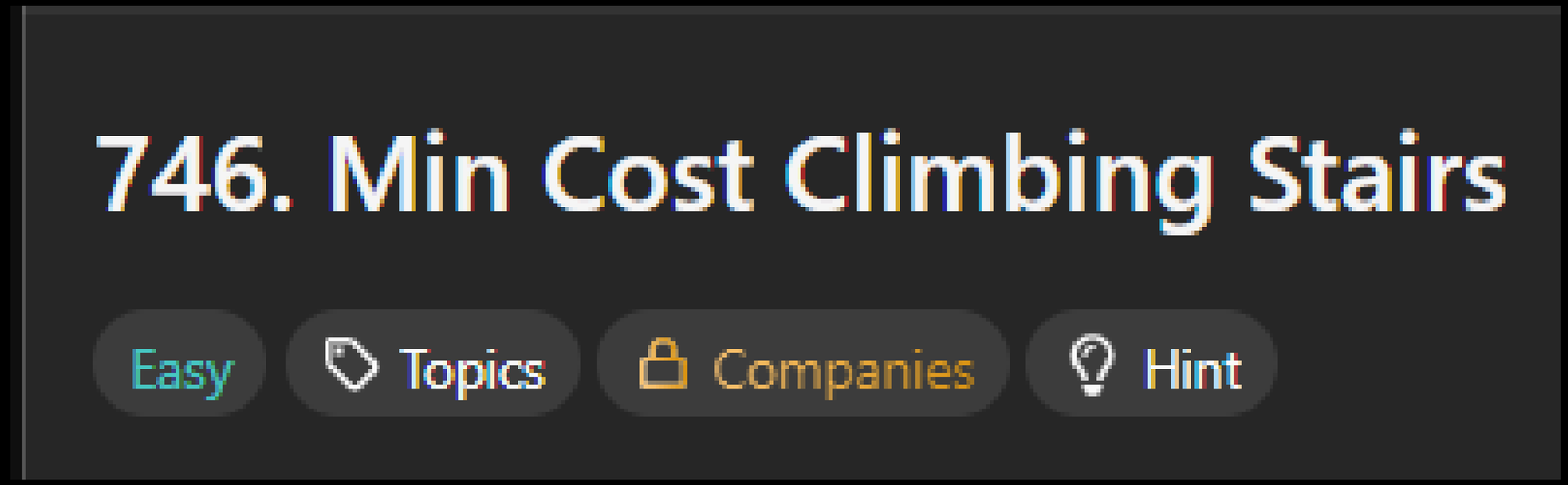
        int[] dp = new int[n + 1];

        dp[1] = 1;
        dp[2] = 2;

        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }
}
```

Let’s modify the problem slightly

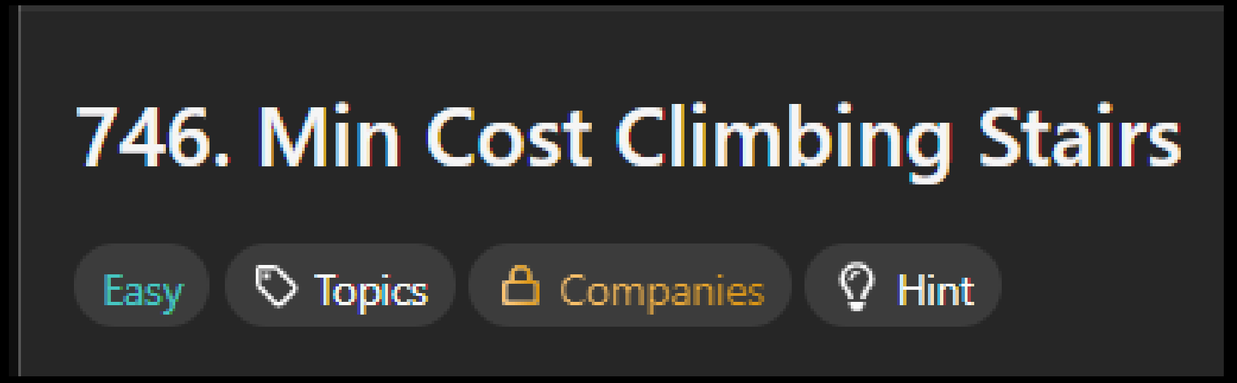


Explanation

The problem structure is almost identical, with one small change:
Each step has a “cost” paid when using the step

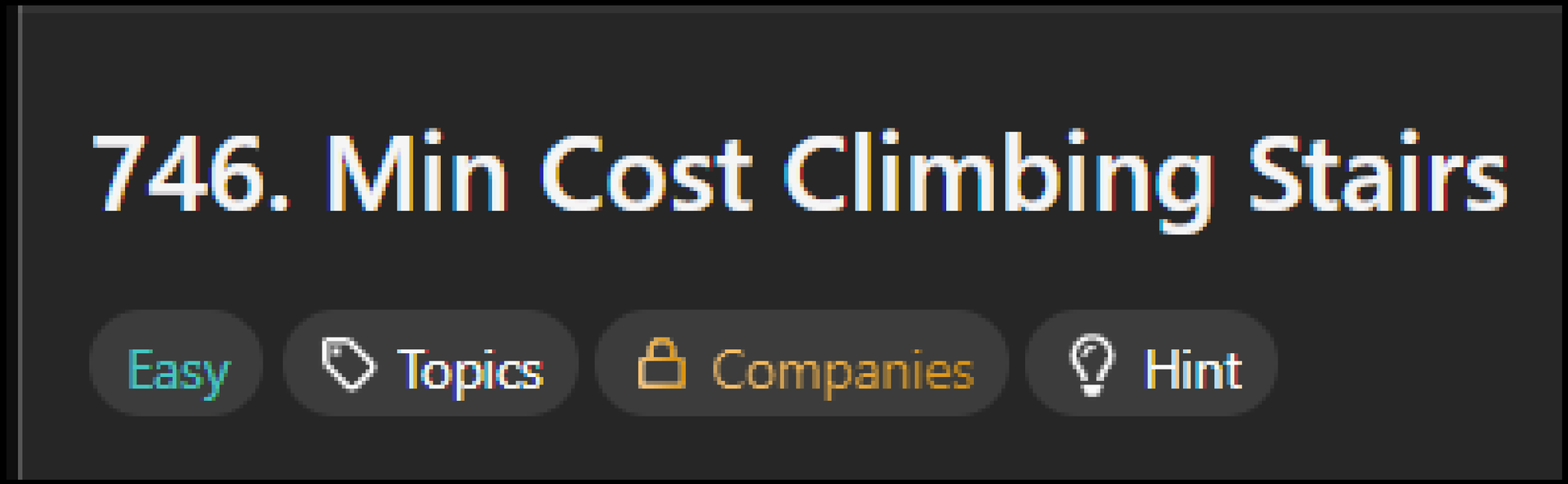
- cost[i] = cost at i-th step

Instead of finding how many paths we can take to the top, we want to find the minimum amount of money we have to pay to get to the top



Go for it!

(And remember, it really isn't too different from the last problem!)



Solution

In the previous prblem, our base cases were:

F(1) = 1
F(2) = 2

In this problem, the base case is provided to us:

You can either start from the step with index `0`, or the step with index `1`.

Constraints:

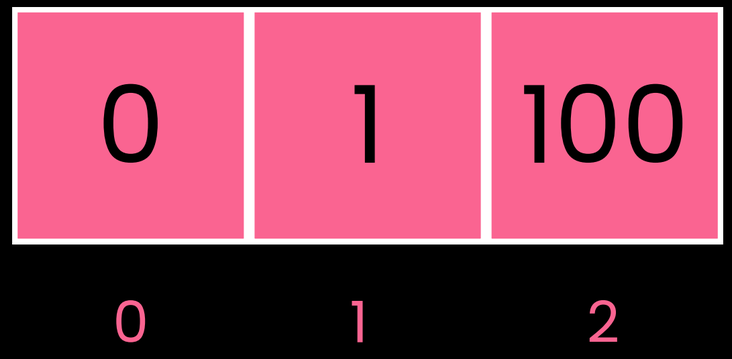
- `2 <= cost.length <= 1000`

This means we will always have the cost of the first two steps

Solution

Input: cost = [1,100,1,1,1,100,1,1,100,1]

Base case:

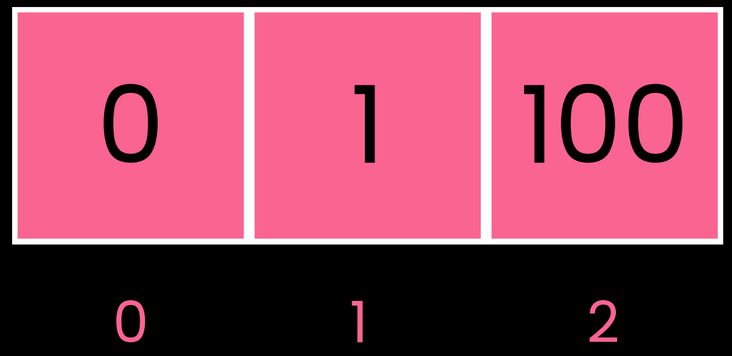


Solution

```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
```

From here, we have two paths to get to step three
Step 1 → Step 3
Step 2 → Step 3

For each one, the total cost we will pay to get to step three is equal to:
Cost of Step 3 + Cost of previous step

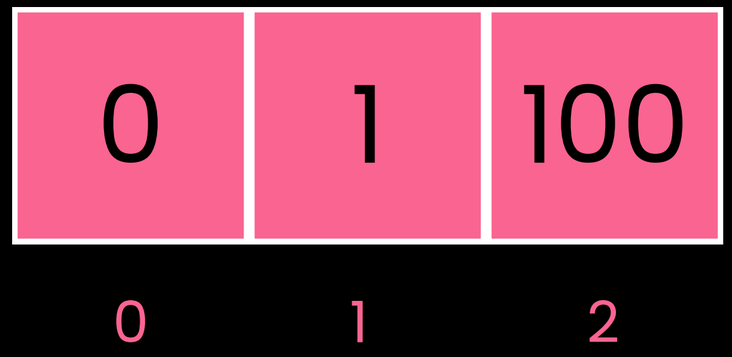


Solution

```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
```

Total Cost of Step 3 = Total cost of Step 1 + Cost of Step 3
or Total cost of Step 2 + Cost of Step 3

Because we want to minimize cost, this becomes:
Total Cost of Step 3 = min(Step 1, Step 2) + Cost of Step 3

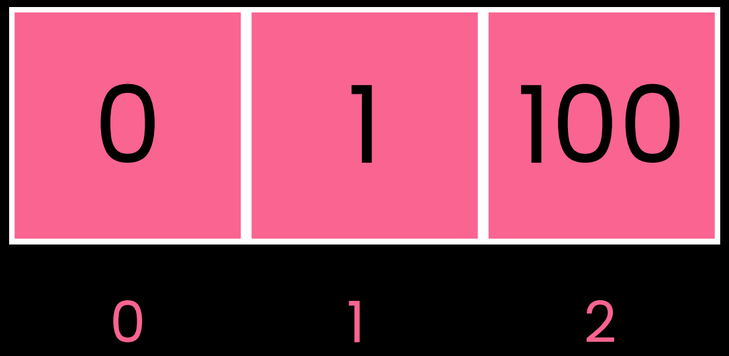


Solution

```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
```

Total Cost of Step 3 = min(Step 1, Step 2) + Cost of Step 3

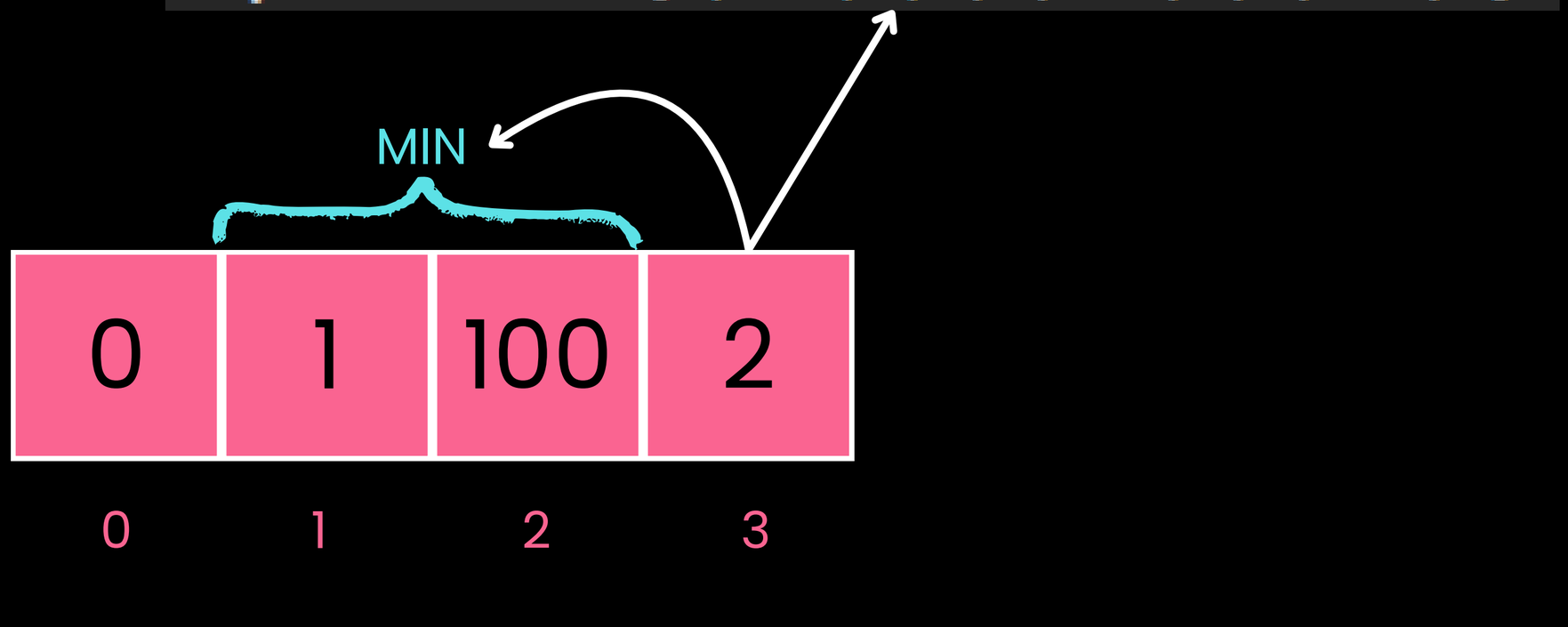
Extrapolating this, we get
Total Cost of step n = min(n-1, n-2) + Cost(n)



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

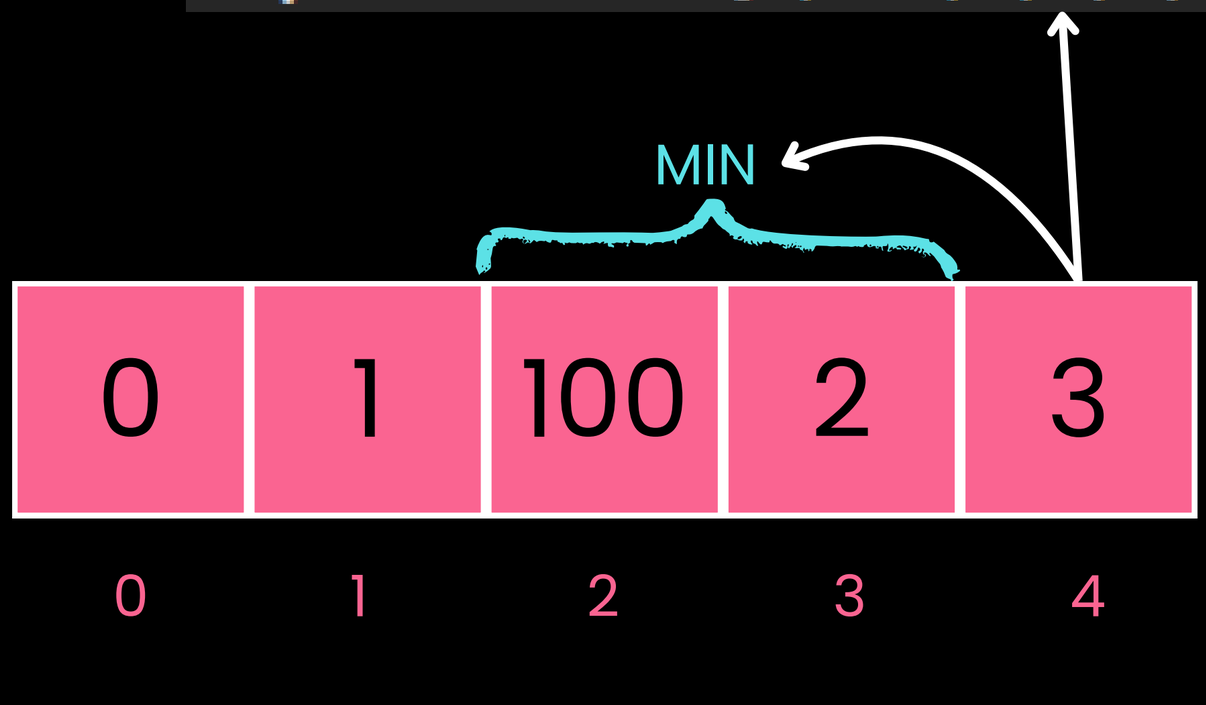
```
Input: cost = [1,100,1,1,1,100,1,1,100,1]
```



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

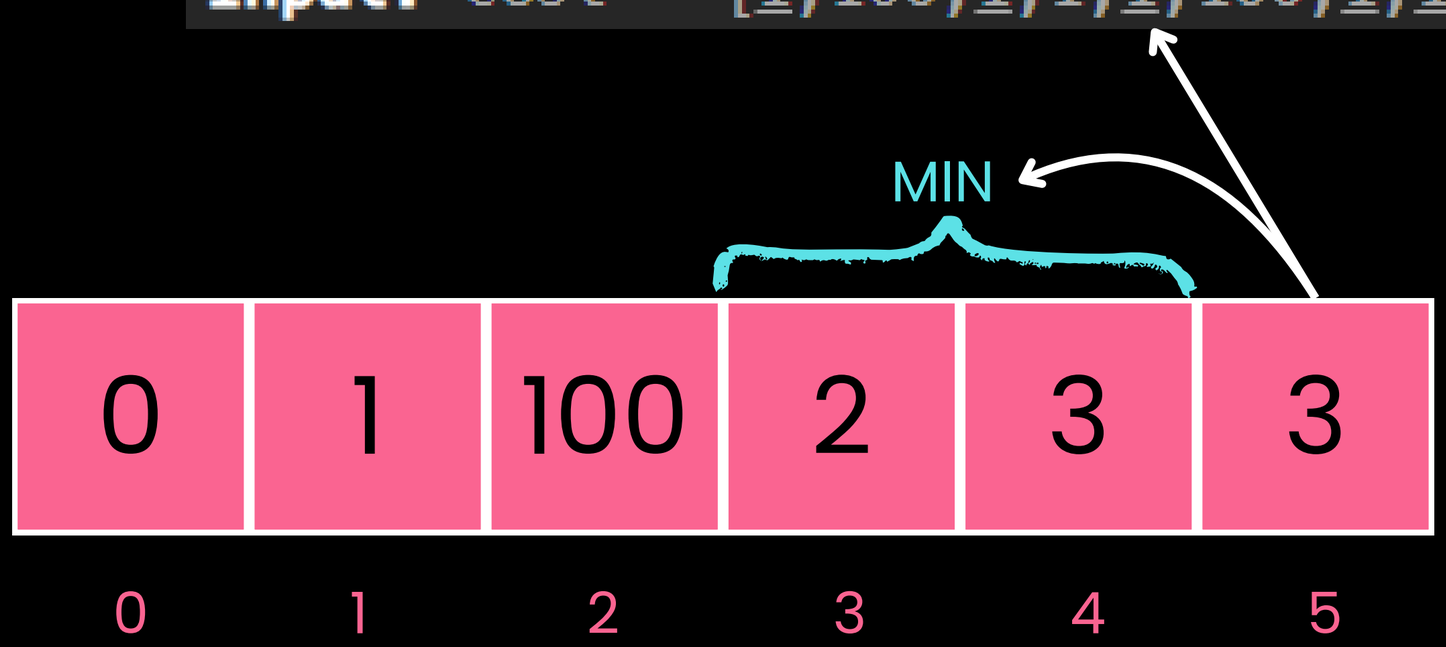
Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

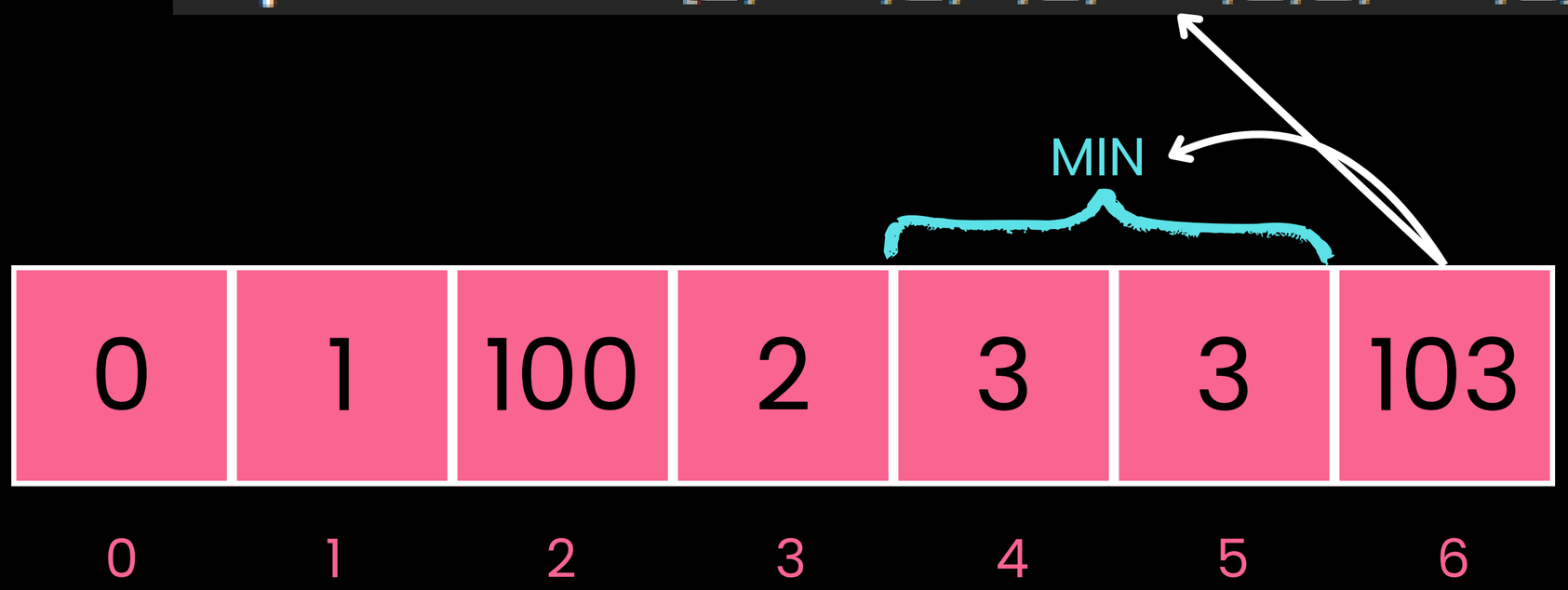
Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

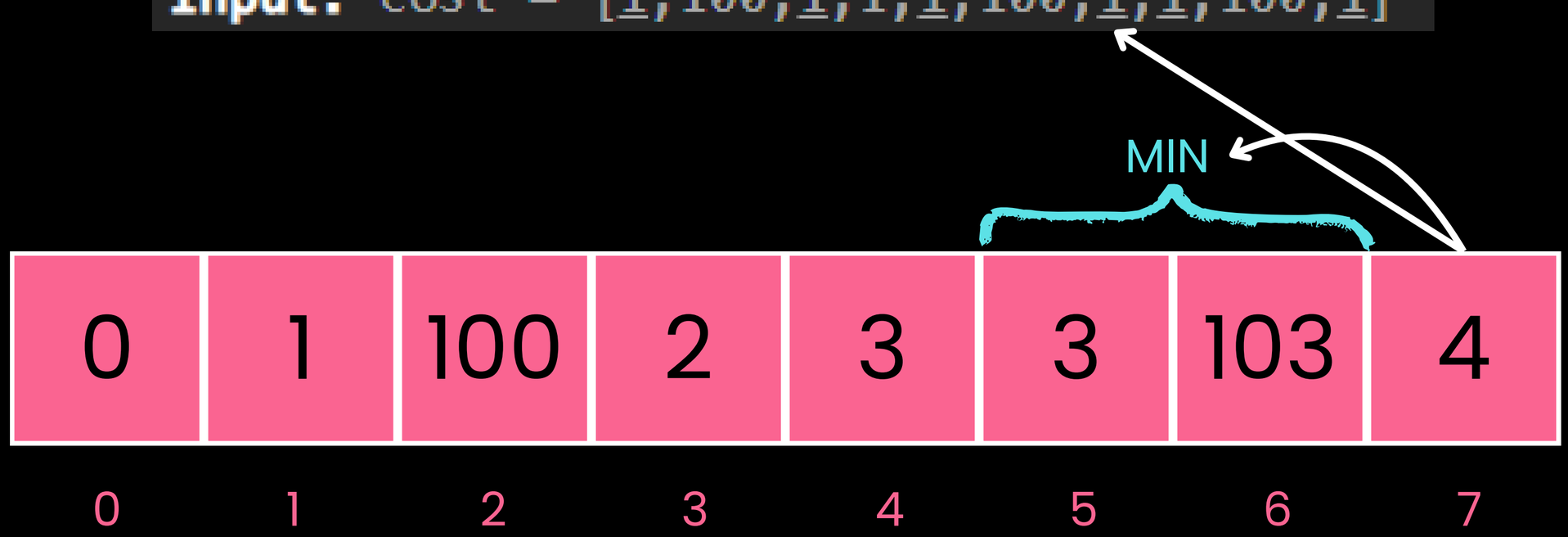
Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

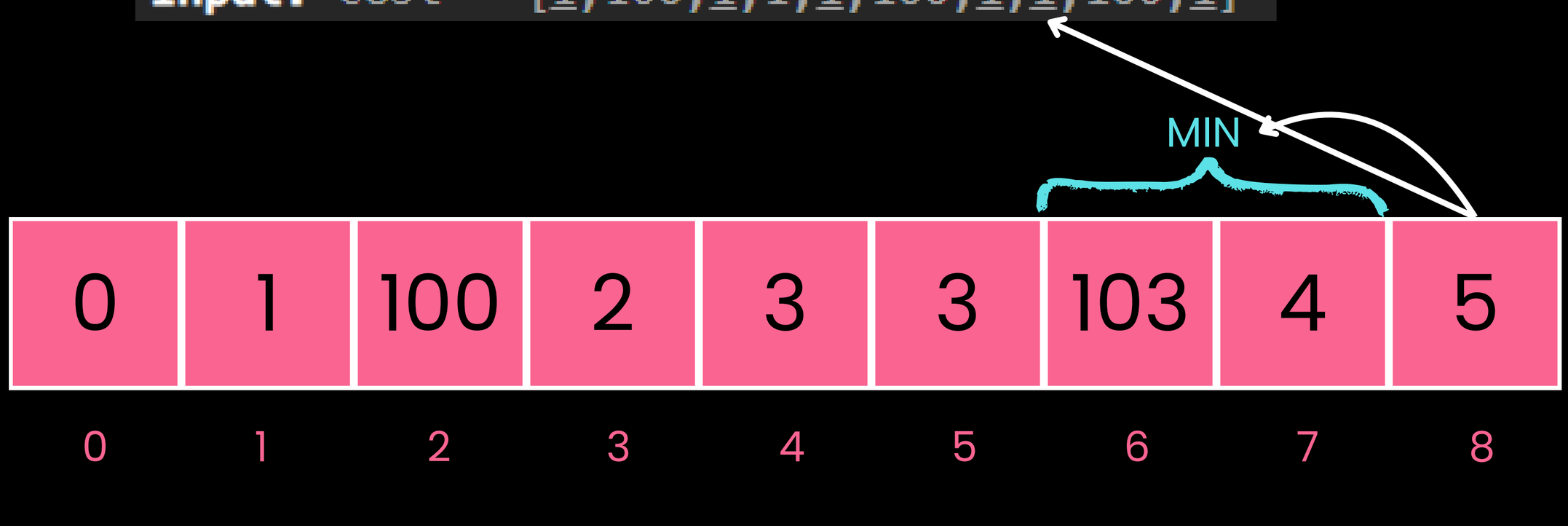
Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

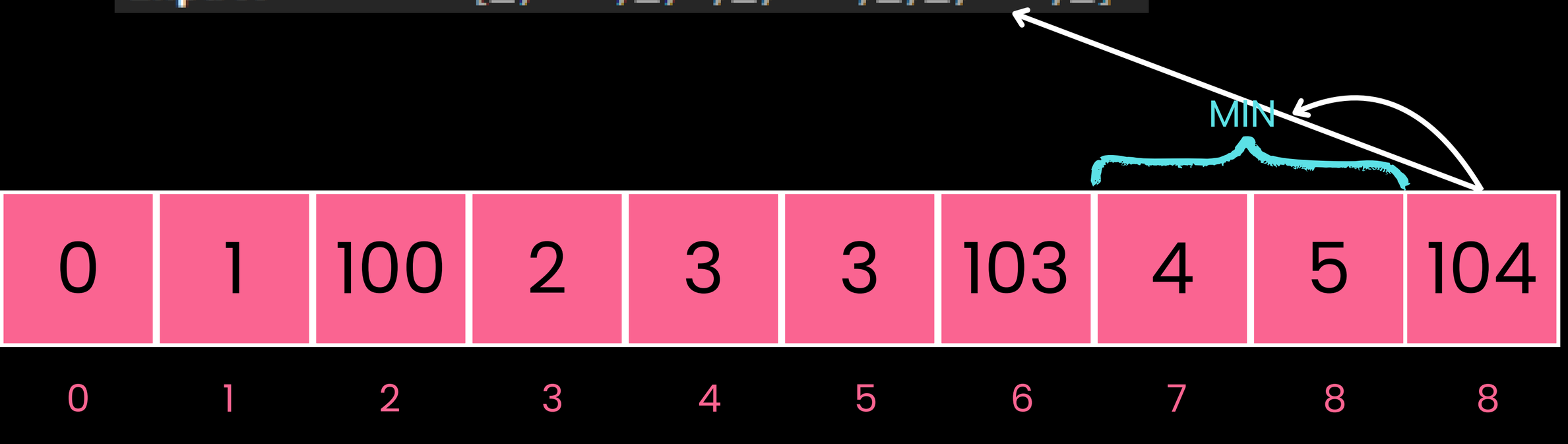
Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

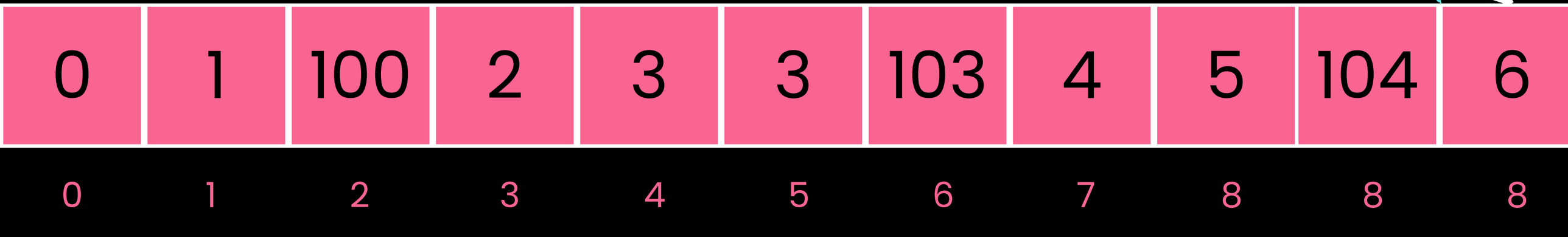
Input: cost = [1,100,1,1,1,100,1,1,100,1]



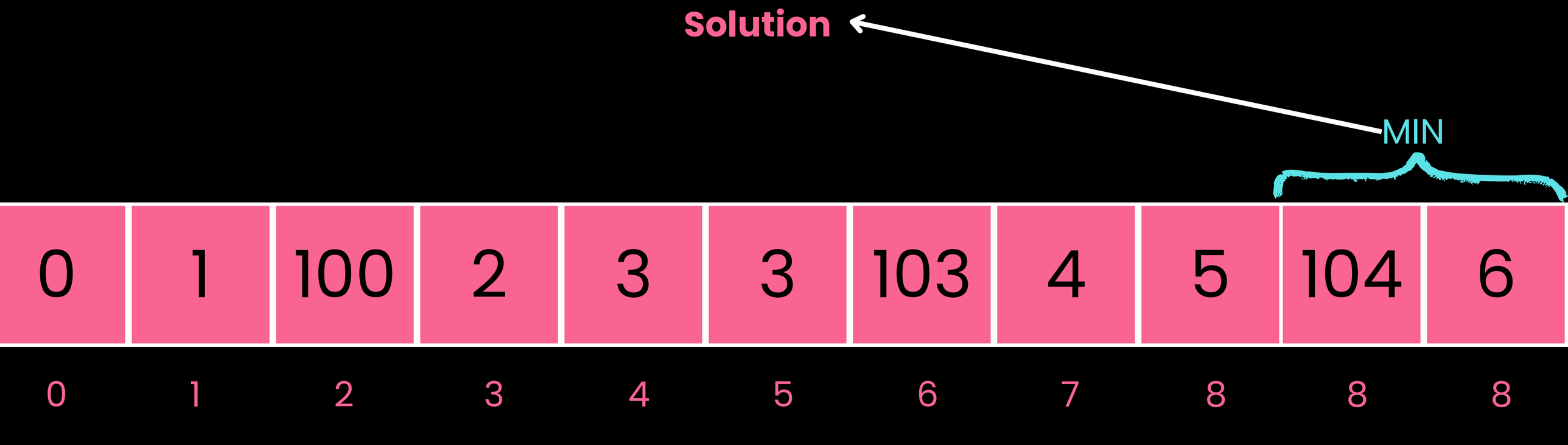
Solution

Total Cost of step n = min(n-1, n-2) + Cost(n)

Input: cost = [1,100,1,1,1,100,1,1,100,1]



Solution



Python

```python
def minCostClimbingStairs(cost: list[int]) -> int:
    n = len(cost)
    dp = [0] * n

    dp[0] = cost[0]
    dp[1] = cost[1]

    for i in range(2, n):
        dp[i] = min(dp[i-1], dp[i-2]) + cost[i]

    return min(dp[n-1], dp[n-2])
```

Java

```java
class Solution {
    public int minCostClimbingStairs(int[] cost) {
        int n = cost.length;
        int[] dp = new int[n];

        dp[0] = cost[0];
        dp[1] = cost[1];

        for (int i = 2; i < n; i++) {
            dp[i] = Math.min(dp[i-1], dp[i-2]) + cost[i];
        }

        return Math.min(dp[n-1], dp[n-2]);
    }
}
```