🐆

Explanation

🔍 Goal

We are counting **how many square submatrices with all 1s** exist in a given binary matrix.

Let's say you're at cell `(i, j)` and `matrix[i][j] == 1`. You want to know:

- What is the **maximum size of square** (ending at `(i, j)`) where **all cells are 1s**?

🧠 Core Idea (Dynamic Programming Table)

We define:

```
dp[i][j] = size of the largest square that ends at (i, j)
```

That means:

- If `dp[i][j] = 2`, it means there is a 2×2 square ending at that cell.
- If `dp[i][j] = 1`, it means only a 1×1 square ends at that cell.

We'll build this table based on previous results.

🧱 Transition Logic

If `matrix[i][j] == 0`, clearly, no square can end here. So `dp[i][j] = 0`.

But if `matrix[i][j] == 1`, we **look in 3 directions**:

- **Left** → `dp[i][j-1]`
- **Up** → `dp[i-1][j]`

- **Top-left (diagonal)** → `dp[i-1][j-1]`

Why? Because:

> To form a square of size k ending at (i, j), you need a square of size k-1 ending at all 3 of those neighbors.

🔁 So we take:

```
dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
```

This ensures we only count the largest square that can be extended from all 3 directions.

📊 Example Visualization

For this input:

```
[
  [0,1,1,1],
  [1,1,1,1],
  [0,1,1,1]
]
```

We compute the `dp` table:

```
[
  [0,1,1,1],
  [1,1,2,2],
  [0,1,2,3]
```

```
]
```

How to interpret this?

Each value at `(i, j)` tells you:

> Number of largest square that ends at that cell.

If `dp[i][j] = 3`, it means there's:

- One 3×3 square
- Embedded inside it: one 2×2 square
- And one 1×1 square

But we only count the 3×3 once **per cell**. To get the total number of squares, we **sum** all `dp[i][j]`.

✅ Base Case

We initialize the first row and column separately:

```
dp[i][0] = matrix[i][0]  # only possible square is 1×1
dp[0][j] = matrix[0][j]  # only possible square is 1×1
```

🧮 Final Answer

Add all the values in `dp` → gives total number of square submatrices with all 1s.

🕒 Time and Space Complexity

- **Time:** `O(m * n)`

- **Space:** `O(m * n)` → can be optimized to `O(n)` since we only need the previous row.

How is DP understanding that, like, it can take the whole square? What if it turns out to be a rectangle?

Let’s take your example:

```
[1, 1]
[1, _]  ← let’s figure out what goes in the `_`
```

We're trying to compute `dp[1][1]`. So let's walk through it **like how the code does**, and also **intuitively explain** why the `min()` ensures **we always build a **square**, not a rectangle.

Step-by-Step with DP

We are at cell `(1, 1)` in the matrix.

Let’s write down the values around it:

```
dp[0][0] = ?  ← diagonal (top-left)
dp[0][1] = ?  ← up
dp[1][0] = ?  ← left
```

Now, assume we already computed the `dp` values for those cells based on the `matrix`. Suppose all those cells are `1` in the matrix, so their `dp` values will be:

```
dp[0][0] = 1
dp[0][1] = 1
```

```
dp[1][0] = 1
```

Then:

```
dp[1][1] = 1 + min(1, 1, 1) = 2
```

So now:

```
matrix:
[1, 1]
[1, 1]

dp:
[1, 1]
[1, 2]
```

This means: a square of size 2×2 ends at `(1,1)` ✅

💡 Why `min()` Guarantees a Square (Not a Rectangle)

We use:

```
dp[i][j] = 1 + min(
    dp[i-1][j],     # top
    dp[i][j-1],     # left
    dp[i-1][j-1]    # top-left
)
```

Why all 3?

Let's say one of them is **smaller** (say 0). **That means the square in that direction is not big enough, so we cannot extend into a bigger square.**

Let's take a **counterexample** where it could be a rectangle:

```
matrix:
[1, 1]
[0, 1]
```

So the cell `(1, 1)` is 1, but:

- `matrix[1][0] = 0` → so `dp[1][0] = 0`

Then:

```
dp[1][1] = 1 + min(dp[1][0], dp[0][1], dp[0][0])
         = 1 + min(0, 1, 1) = 1
```

Which means: **we can only form a 1×1 square** at `(1,1)`, even though **visually** there's a "row of 1s" or a "column of 1s". The `min()` ensures **all sides must be part of a valid square**.

✔️ So `min()` acts as a gatekeeper:

It says: "You can only grow the square if **all 3** neighbors support it."

If one of them is small (say just a row), then we can't grow a square, only rectangles — and we don't count those.

🔑 Intuition Summary

- `dp[i][j] = 1 + min(top, left, top-left)` makes sure we **only count squares**, not rectangles.
- All three directions must have at least `k-1` sized squares to grow a `k x k` square.
- The value at `dp[i][j]` tells you the **largest square** that ends at that cell.

- The final result is the **sum of all such square sizes**.
