2579. Count Total Number of Colored Cells

Medium Math

Problem Description

In this problem, we are presented with an infinite two-dimensional grid made up of unit cells. Initially, all cells are uncolored. We are tasked with executing a given routine over the course of `n` minutes, with each minute corresponding to a step in the coloring process:

- On the first minute, we have to color one arbitrary cell blue. This can be any cell in the grid.
- For every subsequent minute, every uncolored cell that is adjacent to at least one blue cell must be colored blue.

The objective is to return the total number of cells that have been colored blue at the end of `n` minutes.

Since the grid is infinite, we are not constrained by edges and every minute will see an expansion of the blue region. Visually, the blue region forms a diamond-like shape that grows each minute.

Intuition

Upon analyzing the pattern of growth, we notice that with each passing minute, the number of colored cells increases in a predictable way. The blue region expands symmetrically each minute, adding a "layer" of cells.

- At minute 1, we have 1 blue cell.
- At minute 2, we add 4 more blue cells around the original, totaling 5 blue cells.
- At minute 3, we add 8 more blue cells surrounding those, totaling 13 blue cells.

Continuing this pattern, we realize that the number of blue cells added each minute is always an even number and the total number of blue cells forms an arithmetic progression.

To derive a formula, we observe:

- The total number of columns of cells that are colored is `2 * n - 1`. This is because for each minute after the first, we add two more columns (one on each side).
- The number of blue cells in these columns forms two sequences of odd numbers, starting with 1 and increasing by 2 each time: `1, 3, 5, ..., 2 * n - 1, ..., 5, 3, 1`.

To find the sum of such a pattern, we notice that the total can be broken down into three parts:

- The sum of the first half which is an arithmetic progression: `1, 3, 5, ..., 2 * n - 3`
- The single middle term which is the largest: `2 * n - 1`
- The sum of the second half which mirrors the first: `..., 5, 3, 1`

By calculating the sum of the arithmetic progression for the first half and then doubling it (to account for the second half) and lastly adding the middle term, we arrive at the total number of blue cells at the end of `n` minutes.

The sum of the first half is computed using the formula for the sum of an arithmetic progression: `n/2 * (first term + last term)`. Here, the `n/2` is actually `(n - 1)` because we are not including the largest term of `2 * n - 1`. So, the sum of the first half is `(n - 1) / 2 * (1 + (2 * n - 3))`. This simplifies to `(n - 1) * n - 1`.

Since the second half is identical to the first, we double this sum and add the middle term:

```
2 * (sum of first half) + (middle term)
2 * ( n * (n - 1) ) + 1
2 * n * (n - 1) + 1
```

And that's the total number of colored cells after `n` minutes, which is the solution provided by the code snippet given.

Solution Approach

The problem we're dealing with has a straightforward solution once the pattern of the growth of blue cells is identified. Since we're not actually coloring cells on a grid but rather calculating the number of cells that would be colored, we can use a mathematical approach to determine the solution without any complex algorithm or data structure. In particular, the problem can be solved with a simple formula derived from the observation of the growth pattern.

As mentioned in the intuition behind the solution, after the `n`th minute, we have a diamond-like shape consisting of `2 * n - 1` columns, with the count of blue cells in each column represented by the sequence:

```
1, 3, 5, ..., 2 * n - 1, ..., 5, 3, 1
```

We recognized this pattern as two identical arithmetic progressions with an additional single term (`2 * n - 1`) in the middle. As no complex data structure is needed to represent the pattern, the implementation focuses on deriving the formula to calculate this sum directly.

The solution method outlined in the Reference Solution Approach uses the properties of an arithmetic progression—the sum of an arithmetic series can be calculated by multiplying the average of the first and last terms by the number of terms—along with the symmetry of the pattern of the blue cells.

Here's an explanation of the formula based on the provided solution:

- Since we have two identical arithmetic progressions and a single middle term, calculating the sum of one progression and doubling it is more efficient than summing all terms individually.
- The first progression summation is the sum of the first `n-1` odd numbers (since the `n`th term, `2 * n - 1`, is considered separately).

The formula derived for the sum of the first `n-1` terms of this progression is `n * (n - 1)`, because:

- There are `n-1` terms.
- The average value of the terms is `(first term + last term) / 2`, which for our sequence is `(1 + (2 * n - 3)) / 2` (this simplifies to `n - 1`).
- Thus, `n-1` terms each with an average value of `n - 1` is `(n - 1) * (n - 1)` or `n * (n - 1)` without the minus 1 which was the single center column.

We then multiply by 2 for the two progressions and add the single middle term: `2 * n * (n - 1) + 1`

Thus, the implementation `return 2 * n * (n - 1) + 1` directly computes the final number of blue cells without having to simulate the actual process, making it a very efficient solution that runs in constant time O(1), since no loops or iterations are required.

Example Walkthrough

To clearly understand the given problem and the proposed solution approach, let's walk through a small example. Assume we are interested in finding out how many cells will be blue after `n = 4` minutes.

Here are the steps that will occur over the 4 minutes:

1. **Minute 1:** Start by coloring one arbitrary cell blue. The total number of blue cells is now 1.
2. **Minute 2:** Color every uncolored cell adjacent to a blue cell. This step adds 4 blue cells surrounding the original one, bringing the total to 5.
3. **Minute 3:** Repeat the process, adding 8 more blue cells around the ones from minute 2. The new total is 13 blue cells.
4. **Minute 4:** A similar expansion happens, adding 12 more blue cells, making the total number of blue cells 25.

At this point, we have the following structure for the blue cells:

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

As we can see, the pattern forms a diamond shape. We can calculate the total number of blue cells by adding the ones in each row:

- 1 cell in the first and last rows
- 3 cells in the second and second-to-last rows
- 5 cells in the third and third-to-last rows
- 7 cells in the fourth row (the middle row)

Now, rather than summing these individually, we'll apply the solution approach. We know the number of columns of blue cells is `2 * n - 1`, which for `n = 4` is `7`. We can calculate the total number of blue cells by considering the sum of an arithmetic progression with `(n-1)` terms (where `n` is the number of minutes) and then doubling it and adding the largest term, which is the number of cells in the middle row, `2 * n - 1`.

- The arithmetic progression (excluding the middle term) is: `1, 3, 5`, and we thus have `n - 1 = 3` terms. The sum of these terms is `1 + 3 + 5 = 9`.
- The middle term (the number of cells in the middle row) when `n = 4` is `2 * 4 - 1 = 7`.
- Then, following the formula, we calculate: `2 * (sum of first half) + (middle term)`.
- Substituting the values, we get `2 * 9 + 7 = 18 + 7 = 25`.

The formula computed the same number of blue cells that we counted manually (25). This verifies our solution approach.

Using the derived formula, we can generalize for any `n`:

```
return 2 * n * (n - 1) + 1
```

For `n = 4`, it simplifies to `2 * 4 * (4 - 1) + 1 = 2 * 4 * 3 + 1 = 24 + 1 = 25`. This confirms the correctness of the implementation for the example of `n = 4` minutes.

Solution Implementation

Python

```python
class Solution:
    def coloredCells(self, n: int) -> int:
        # The formula calculates the number of colored cells in a particular pattern,
        # where n represents the size of the grid.
        # It is based on a mathematical observation that with each increase in n,
        # there are 2*(n-1) additional colored cells from the previous step,
        # plus the center cell which is always colored.

        # Calculate the number of colored cells using the given formula:
        # 2 * n * (n - 1) is the pattern growth as we move to larger grids,
        # +1 accounts for the center cell, which is always colored.
        return 2 * n * (n - 1) + 1
```

Java

```java
class Solution {
    // Method to calculate the number of colored cells in an n x n grid
    // following a specific pattern.
    public long coloredCells(int n) {
        // The formula to calculate the colored cells is based on the observation
        // that there are 2 * n * (n - 1) cells that are colored in rows and columns,
        // plus 1 for the initial cell (the center cell in case of an odd-sized grid,
        // or one of the centers in an even-sized grid).

        // 2 * n * (n - 1) accounts for the rows and columns colored cells
        // +1 represents the initial cell that gets colored
        return 2L * n * (n - 1) + 1;
    }
}
```

C++

```cpp
class Solution {
public:
    // Method to calculate the number of colored cells in a pattern
    // Assumes a grid of n by n cells where certain cells are colored
    // based on a given pattern. The pattern rule isn't specified,
    // but the formula 2*n*(n-1) + 1 is applied to determine the result.
    long long coloredCells(int gridSize) {
        // Calculate the number of colored cells according to the given pattern
        // The pattern results in a calculation involving the grid size;
        // 2 multiplied by the grid size (n) and then multiplied by (grid size (n) - 1)
        // Plus 1 for the center or a start cell according to the pattern
        long long numberOfColoredCells = 2LL * gridSize * (gridSize - 1) + 1;

        // Return the calculated number of colored cells
        return numberOfColoredCells;
    }
};
```

TypeScript

```typescript
// This function calculates the number of cells with distinct colors in a pattern.
// The pattern consists of a square grid that starts with a single colored cell at the center,
// and additional cells are colored in concentric square rings outward from the center.
// The input 'n' represents the number of rings around the center cell.
// The output is the total number of colored cells.
function coloredCells(n: number): number {
    // The formula 2 * n * (n - 1) + 1 calculates the total number of colored cells.
    // 2 * n * (n - 1) computes the number of cells in the outer rings, since each
    // ring contributes n-1 new cells on each of its 4 sides, minus the 4 corners which are
    // counted twice, hence 4 * (n - 1) - 4 simplifies to 4n - 4 - 4 which is 4n - 8.
    // Since each new ring is computed twice due to the overlap we use 2 * n * (n - 1).
    // The + 1 accounts for the starting centered cell which is also colored.
    return 2 * n * (n - 1) + 1;
}

class Solution:
    def coloredCells(self, n: int) -> int:
        # The formula calculates the number of colored cells in a particular pattern,
        # where n represents the size of the grid.
        # It is based on a mathematical observation that with each increase in n,
        # there are 2*(n-1) additional colored cells from the previous step,
        # plus the center cell which is always colored.

        # Calculate the number of colored cells using the given formula:
        # 2 * n * (n - 1) is the pattern growth as we move to larger grids,
        # +1 accounts for the center cell, which is always colored.
        return 2 * n * (n - 1) + 1
```

Time and Space Complexity

The time complexity of the given code is `O(1)`. This is because the calculation of the expression `2 * n * (n - 1) + 1` involves a constant number of mathematical operations that do not depend on the size of `n`. Regardless of the value of `n`, the number of operations remains fixed.

The space complexity of this code is also `O(1)`. There are no data structures (like arrays or lists) that grow with the input size. The only extra memory that is used is for the input `n` and the computed result, both of which take constant space.