319. Bulb Switcher

Medium Brainteaser Math

Leetcode Link

Problem Description

In this problem, there are `n` light bulbs aligned in a row, all initially turned off. The task involves performing a series of `n` rounds. In each round `i`, every `i-th` bulb's state is toggled (i.e., if the bulb is off, it is turned on, and if it is on, it is turned off). The first round involves toggling every bulb, so all bulbs are turned on. The second round toggles every second bulb, turning off all even-numbered bulbs. As the rounds progress, the toggling of bulbs continues where in each round `i`, only the bulbs that are multiples of `i` have their states toggled. The final task is to determine how many bulbs remain on after all `n` rounds have been completed.

Intuition

Upon observing the pattern of toggling, we can notice that a bulb's final state (on or off) depends on how many times it is toggled. Bulbs are toggled only when their positions are factors of the round number. So, the crucial insight is that a bulb ends up being on if and only if it has an odd number of factors. Mathematically, the only numbers that have an odd number of factors are perfect squares because factors usually come in pairs, and a square has a middle factor that is counted only once. For example, 9 is toggled on rounds 1, 3, and 9.

Given this, the problem simplifies to finding out how many perfect squares are there up to `n`, because these will be the bulbs that remain on. The number of perfect squares up to `n` is simply the largest integer square root of `n`, because for each number `x` such that `x^2 <= n`, there is a corresponding perfect square. Consequently, the solution is to calculate the square root of `n` and truncate it to an integer value, which is what the given Python function does by computing `int(n ** (1 / 2))`.

Solution Approach

The implementation of the solution for this problem is surprisingly straightforward due to the recognition of the underlying mathematical pattern. No complex algorithms or data structures are required.

Here's the step-by-step approach, following the pattern discovered:

1. **Identify the pattern:** We observe that each bulb's state is toggled whenever the round number is a factor of the bulb's position. A bulb ultimately remains on only if it's toggled an odd number of times, which corresponds to it having an odd number of factors. This is uniquely characteristic of perfect squares.

2. **Connect to perfect squares:** Since only perfect squares have an odd number of factors (with the square root being a factor that only contributes one to the count because it is equal to its pair), we only need to count the number of perfect squares up to `n`.

3. **Calculate the number of perfect squares:** To find out how many perfect squares there are up to `n`, we take the square root of `n`, as this gives us the largest number whose square is less than or equal to `n`. For instance, if `n = 16`, the square root is `4`, and there are 4 perfect squares (1, 4, 9, and 16).

4. **Implement the solution:** In the given Python code, the function `bulbSwitch` takes an integer `n` and returns `int(n ** (1 / 2))`. `n ** (1 / 2)` computes the square root of `n`, and the `int()` function truncates the result to the largest integer less than or equal to the square root, effectively counting the number of perfect squares up to `n`, which is the number of bulbs that remain on after `n` rounds.

This intuitively simple yet effective approach takes advantage of the inherent mathematical properties of the problem, resulting in a time complexity of O(1) since it is a direct mathematical calculation, and a space complexity of O(1) because no additional space is required regardless of the value of `n`.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose there are `n = 10` light bulbs in a row. We need to find out how many bulbs will be on after performing the 10 rounds of toggling defined in the problem.

1. **Initial State**: All bulbs are off.
   ```
   1  [off, off, off, off, off, off, off, off, off, off]
   ```

2. **After Round 1**: Every bulb is toggled (turning all bulbs on).
   ```
   1  [on, on, on, on, on, on, on, on, on, on]
   ```

3. **After Round 2**: Every 2nd bulb is toggled (turning every even-positioned bulb off).
   ```
   1  [on, off, on, off, on, off, on, off, on, off]
   ```

4. **After Round 3**: Every 3rd bulb is toggled.
   ```
   1  [on, off, off, off, on, on, on, off, off, off]
   ```

5. Continuing this sequence of toggling every i-th bulb for rounds 4 through 10, we start to observe the pattern the problem description pointed out. Let's jump to the conclusion using the insight that only bulbs at perfect square positions will remain on:

- Bulb 1 is toggled in round 1 (perfect square: 1^2)
- Bulb 4 is toggled in rounds 1 and 2 (perfect square: 2^2)
- Bulb 9 is toggled in rounds 1, 3 (perfect square: 3^2)

6. No other bulbs up to bulb 10 are at perfect square positions, meaning all bulbs beyond bulb 9 will be off at the end of 10 rounds.

7. We use the solution approach, which tells us to find the largest integer square root of `n` (`n = 10`), which is `int(10 ** (1 / 2)) = 3`. There are 3 perfect squares less than or equal to 10: 1, 4, and 9.

8. **Final State**: Three bulbs remain on - bulbs 1, 4, and 9.

   ```
   1  [on, off, off, on, off, off, off, off, on, off]
   ```

There are 3 bulbs on after 10 rounds. This matches the result of our direct calculation using the square root, solidifying the intuition and solution approach presented earlier.

Python Solution

```python
class Solution:
    def bulbSwitch(self, n: int) -> int:
        # Calculate the square root of n, which corresponds to the
        # number of bulbs that will be on after n rounds of toggling
        # since only perfect square positions will be toggled an odd
        # number of times and hence end up being on.
        # The int() function takes the floor of the square root to
        # return how many perfect squares are there up to and including n.

        return int(n ** 0.5)

```

Java Solution

```java
class Solution {
    // Method to find out how many bulbs are left switched on after n rounds
    public int bulbSwitch(int n) {
        // The bulbs that remain on are the ones whose indices are perfect squares
        // because they have an odd number of factors/divisors (e.g. 1, 4, 9, 16, ...).
        // The square root of 'n' gives the largest integer 'k' such that k^2 is less than
        // or equal to 'n', which is the number of perfect squares in the range [1...n].
        // Thus, it also represents the number of bulbs that will remain on.

        // Calculate the square root of 'n' and cast it to an integer since the bulbs are countable (integral).
        return (int) Math.sqrt(n);
    }
}

```

C++ Solution

```cpp
#include <cmath>

class Solution {
public:
    // Method to find out how many bulbs are left switched on after n rounds
    int bulbSwitch(int n) {
        // The bulbs that remain on are the ones whose indices are perfect squares.
        // They have an odd number of factors (e.g., 1, 4, 9, 16, ...).
        // The square root of 'n' provides the largest integer 'k'
        // such that k^2 is less than or equal to 'n'.
        // This signifies the number of perfect squares in the range [1, n]
        // and consequently, the number of bulbs that will remain on.

        // Calculate the square root of 'n' and cast it to an integer as the bulbs
        // are countable (integral) entities.
        return static_cast<int>(sqrt(n));
    }
};

```

Typescript Solution

```typescript
// Function to determine how many bulbs are left switched on after n rounds
function bulbSwitch(n: number): number {
    // The bulbs that remain on are those whose indices are perfect
    // squares because they have an odd number of factors (e.g., 1, 4, 9, 16,...).
    // The largest perfect square that is less than or equal to 'n' is determined
    // by the square root of 'n'. The floor of this square root will give us the
    // count of the perfect squares and also the number of bulbs that remain on.

    // Calculate the square root of 'n', and use Math.floor to get the largest
    // integer less than or equal to the square root of 'n'
    return Math.floor(Math.sqrt(n));
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(1)` because the computation performed is a square root operation, which is done in constant time regardless of the size of the input `n`.

Space Complexity

The space complexity of the function is also `O(1)` since it only uses a fixed amount of space to store the intermediate results and the final output without any dependence on the input size.