# 1952. Three Divisors

Easy Math

Leetcode Link

## Problem Description

The problem requires determining whether a given integer `n` has exactly three positive divisors. An integer `m` is considered a divisor of `n` if there exists another integer `k` such that `n = k * m`. For `n` to have exactly three positive divisors, it must have 1 and itself as divisors, and exactly one other divisor in between.

## Intuition

The intuition behind the solution is based on the properties of numbers with exactly three divisors. In such cases, `n` would have to be a square of a prime number since the divisors of `n` would then be 1, the prime number itself, and the square of the prime (which is `n`). This is because prime numbers only have two divisors: 1 and themselves. When squared, they introduce exactly one new divisor, the square itself.

The solution approach starts by iterating through possible divisors starting from 1 and going up to the square root of `n`. For each potential divisor `i`, the code checks if `i` divides `n` evenly (i.e., `n % i == 0`). If it does, it increments a counter. Special attention is needed when `i` and `n/i` are the same, which occurs if `n` is a perfect square. In this special case, the counter should only be increased by 1 to avoid double-counting the divisor.

If the total count becomes 3, it means we've found exactly two divisors (other than 1 and `n` itself). The loop can stop at the square root of `n` because if `n` has a divisor greater than its square root, then it must also have a corresponding divisor smaller than the square root (since `n` can be factored into a product of two numbers, one of which must be less than or equal to the square root). Hence, any number with more than one divisor apart from 1 below its square root would have more than three positive divisors overall.

The function `isThree` returns `True` if the counter equals 3, indicating `n` has exactly three positive divisors. Otherwise, it returns `False`.

## Solution Approach

The solution employs a simple but effective algorithm to determine if the given number `n` has exactly three divisors. The primary data structure used is a simple integer variable `cnt` that is used to keep track of the count of divisors found.

Here's the step-by-step breakdown of how the implementation works:

- Initialize a counter `cnt` to zero. This will keep track of the number of divisors of `n`.
- Start iterating `i` from `1` through to `n`'s square root, which is the maximum possible value for divisors of `n` other than `n` itself. We check up to `n // i` to avoid considering any factors larger than the square root of `n`, since those would indicate that 'n' has more than 3 divisors.
- For each `i`, check if it's a divisor of `n` by using the modulo operation (`n % i == 0`). If it is, `i` is a divisor of `n`.
  - If `i` is equal to `n // i`, increment `cnt` by 1 because `n` is a perfect square and `i` is its square root. We increment by 1 to avoid overcounting the divisor.
  - Otherwise, increment `cnt` by 2, as both `i` and `n // i` are distinct divisors of `n`.
- After the loop, return whether the counter `cnt` equals 3. If `cnt` equals 3, it means that we have found exactly one divisor of `n` other than 1 and `n` itself, which signifies that `n` is a perfect square of a prime number. Thus, `n` would have exactly three positive divisors.

This implementation uses constant space (for the counter) and has a time complexity of O(sqrt(n)), as it only needs to iterate through values up to the square root of `n`.

The use of the square root as an upper bound for the loop is a common optimization technique in algorithms dealing with factors or divisors since the properties of divisors come in pairs; for a given divisor pair `a` and `b` such that `a * b = n`, one of them must be less than or equal to the square root of `n`, and the other must be greater than or equal to it.

No complex data structures or patterns are needed, just careful iteration and checking based on the mathematical properties of divisors.

### Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose we want to check whether the integer `n = 9` has exactly three positive divisors.

1. We initialize a counter `cnt` to zero to keep track of the number of divisors of `n`.

2. We start iterating `i` from `1` through to the square root of `n`. In our case, `n = 9`, so the square root is `3`.

3. We check if `i = 1` is a divisor of `n` using `n % i`. Since `9 % 1 == 0`, `1` is a divisor of `n`. We increment `cnt` by `1` (since the counterpart of `1` is `n` itself, which we don't count now).

4. We then check the next integer `i = 2`. Does `9 % 2 == 0`? No, so `2` is not a divisor of `n`, and we leave `cnt` unchanged.

5. Finally, we check if `i = 3` is a divisor. Since `9 % 3 == 0`, `3` is a divisor of `n`. However, in this case, `i` is equal to `n // i` (since `9 // 3 == 3`), which means `n` is a perfect square and `3` is its square root. Therefore, we increment `cnt` by `1` again, to account separately for the divisor `3` and for the number `9` itself.

6. At this point, we've finished iterating through all the numbers up to the square root of `n`. Our counter `cnt` now equals `2` (as we found that both `1` and `3` are divisors, and we count `n` itself separately).

7. We check if `cnt` equals `3`. As our count is `2`, we can conclude that `n = 9` does indeed have exactly three positive divisors: `1`, `3`, and `9` itself.

8. Hence, the function would return `True` for `n = 9`.

Through this example, we can observe the ease with which we can determine if a number has exactly three divisors by leveraging its prime factorization and the properties of perfect squares.

## Python Solution

```python
class Solution:
    def isThree(self, num: int) -> bool:
        # Initialize a variable to count the number of divisors
        divisor_count = 0

        # Initialize a loop variable starting at 1
        divisor = 1

        # Loop through possible divisors up to the square root of num
        while divisor <= num // divisor:
            # If divisor evenly divides num, increment the divisor count
            if num % divisor == 0:
                # If the divisor squared is num, it should only be counted once
                if divisor == num // divisor:
                    divisor_count += 1
                else:
                    # Otherwise, there are two distinct divisors (divisor and num // divisor)
                    divisor_count += 2
            # Move to the next possible divisor
            divisor += 1

        # Return True if the total count of divisors is exactly 3, which means
        # 'num' is a prime number that has divisors: 1, itself, and one additional number.
        return divisor_count == 3

```

## Java Solution

```java
class Solution {
    // Method to check if the given number has exactly three divisors
    public boolean isThree(int number) {
        int divisorCount = 0; // Initializing count of divisors

        // Loop from 1 to the square root of the number
        for (int i = 1; i <= number / i; ++i) {
            // Check if 'i' is a divisor of 'number'
            if (number % i == 0) {
                // If 'i' is a divisor, increment divisorCount by 1 if 'i' squared is 'number'
                // otherwise increment by 2 to count both 'i' and 'number / i' as divisors
                divisorCount += (number / i == i) ? 1 : 2;
            }
        }

        // The number has exactly three divisors if divisorCount is 3
        return divisorCount == 3;
    }
}

```

## C++ Solution

```cpp
class Solution {
public:
    // Function to check if the given number has exactly three divisors
    bool isThree(int num) {
        int divisorsCount = 0; // Initialize a counter for the number of divisors

        // Loop to count divisors. Loop runs from 1 to sqrt(num) to avoid unnecessary checks
        for (int i = 1; i <= num / i; ++i) {
            // Check if 'i' is a divisor of 'num'
            if (num % i == 0) {
                // Increase divisor count by 1 if 'i' is the square root of 'num' (i.e., a divisor that is counted only once)
                // Otherwise, increase count by 2 for both divisors 'i' and 'num / i'
                divisorsCount += (num / i == i) ? 1 : 2;
            }
        }
        // A number has exactly three divisors if it has one pair of distinct divisors and one repeated (i.e., 1 and the number itsel
        return divisorsCount == 3;
    }
};

```

## Typescript Solution

```typescript
/**
 * Determines if a positive integer `n` has exactly three divisors.
 * A number has exactly three divisors if it is a square of a prime number.
 * @param {number} n - The positive integer to check.
 * @return {boolean} - Returns true if `n` has exactly three divisors; otherwise, false.
 */
function isThree(n: number): boolean {
    let divisorCount: number = 0; // Initialize counter for the number of divisors.

    // Loop through potential divisors from 1 up to the square root of `n`.
    // If `i` is a divisor of `n`, then n/i is also a divisor.
    // This halves the number of iterations needed as divisors come in pairs.
    for (let i: number = 1; i <= n / i; ++i) {
        if (n % i === 0) { // Check if `i` is a divisor of `n`.
            // If `n / i` equals `i`, it's the square root and should only be counted once.
            // Otherwise, increment the counter by 2 to account for both `i` and `n / i`.
            divisorCount += (n / i === i) ? 1 : 2;
        }
    }

    // Check if the total number of divisors is exactly three.
    return divisorCount === 3;
}

// Example usage:
// const result: boolean = isThree(4);
// console.log(result); // Output would be true since 4 has exactly three divisors: 1, 2, and 4.

```

## Time and Space Complexity

The given code is used to determine whether an integer `n` has exactly three positive divisors. To find this out, the code iterates through potential divisors and checks how many divisors `n` has.

### Time Complexity

The time complexity of this code is `O(sqrt(n))`. This is because the loop runs from 1 up to `sqrt(n)`. The condition `i <= n // i` is equivalent to `i * i <= n`, ensuring that we do not check divisors greater than the square root of `n`. When we find a divisor `i`, we add 2 to the count (`cnt`) since `i` and `n // i` are two distinct divisors unless `i` equals `n // i` (which can only happen if `n` is a perfect square), in which case we add 1.

### Space Complexity

The space complexity of this code is `O(1)`. The algorithm uses a fixed number of integer variables that do not depend on the input size. Hence, the space requirement remains constant irrespective of the input `n`.