2481. Minimum Cuts to Divide a Circle

Easy Geometry Math

Problem Description

The LeetCode problem described asks for the smallest number of straight-line cuts needed to divide a circle into `n` equal slices. There are two types of valid cuts:

1. A cut that goes through the center of the circle, and touches two points on the edge.
2. A cut that touches the center of the circle and one point on the edge.

These cuts can create equal-sized slices of the circle. The challenge is to determine the minimum number of such cuts required to get `n` equal slices.

Intuition

The intuition behind the solution to this problem lies in understanding patterns related to circle division and geometric properties.

For even `n`, every cut that goes through the center of the circle divides it into two additional sections. Therefore, it's possible to create `n` sections with `n/2` such cuts because each cut creates two sections simultaneously.

However, for odd `n`, one cut must first create a singular section, and subsequently, every cut through the center adds two more sections. So for odd numbers greater than one, it's not possible to utilize a single cut to add two sections until we've made an initial cut to create the first single section.

To summarize, for an even `n`, we need only `n/2` cuts (as each cut creates two sections), but for an odd `n` (greater than 1), each cut after the first adds two sections, requiring `n` cuts in total (the first cut plus `n/2` cuts to get to `n-1`, which results in `n-1 + 1 = n` cuts).

The given Python function `numberOfCuts(n: int) -> int` considers these geometric properties and uses bitwise operations for efficiency: `n & 1` checks if `n` is odd (since the least significant bit of an odd number is 1) and `n >> 1` divides `n` by 2 using a bit shift to the right.

Solution Approach

The implementation of the solution utilizes a simple conditional check to determine the number of cuts needed based on whether the number of slices `n` is odd or even.

For an odd `n` greater than 1:

- A total of `n` cuts are needed. The first cut creates the first section and each subsequent cut generates two additional sections. Since odd numbers do not divide cleanly into two equal parts, there's no way to create an odd number of equal sections with fewer cuts without breaking the "equal size" rule.
- The algorithm directly returns `n` in such a case without any further operations.

For an even `n`:

- We can create `n` sections by making `n/2` cuts, each of which goes through the center and creates two new edges, resulting in two new sections.
- The bit shift operation `n >> 1` is effectively a fast way to calculate `n/2` because shifting the bits of a number one position to the right divides the number by two. So for even `n`, the algorithm returns `n >> 1`.

The code structure employs bitwise operations, which are a computationally efficient way to perform integer operations such as checking if an integer is odd and dividing an integer by two. By using a bitwise AND operation with 1 (`n & 1`), the code effectively checks if the least significant bit is set, thereby determining if `n` is odd. Similarly, the bitwise right shift operation (`n >> 1`) divides the number by two.

No complex data structures or algorithms are needed because the solution approach relies entirely on these simple arithmetic and bitwise operations, reflecting a deep understanding of the problem's geometric nature and optimizing for computational speed.

Applying this understanding to the given Python function:

```python
class Solution:
    def numberOfCuts(self, n: int) -> int:
        return n if (n > 1 and n & 1) else n >> 1
```

It's evident that the code checks if `n` is greater than 1 and odd; if both conditions are true, it returns `n`. Otherwise, it performs a bitwise right shift to divide `n` by 2, returning `n >> 1`. This concise and efficient implementation gets us the minimum number of cuts needed to divide a circle into `n` equal slices.

Example Walkthrough

Let's assume we want to find the smallest number of straight-line cuts needed to divide a circle into 6 equal slices.

We follow the solution approach and determine whether 6 is an odd or even number. In this case, 6 is even, so we know we can make `n/2` cuts, each through the center, to divide the circle into `n` equal slices.

According to our solution, we apply a bitwise operation to calculate `n/2` by shifting the bits of `n` one position to the right. For our example, this means shifting the bits of 6 to the right:

The binary representation of 6 is `110`. Shifting the bits one position to the right, we drop off the last `0`, resulting in `11`, which is the binary representation of 3.

Therefore, `6 >> 1` equals 3, indicating we need 3 cuts to divide the circle into 6 equal slices.

This can be visualized as follows:

1. The first cut goes through the center, creating 2 slices.
2. The second cut also goes through the center, perpendicular to the first, creating a total of 4 slices.
3. The third cut, again through the center, must be at a 45-degree angle to the other cuts to create the total of 6 equal slices.

Each of these cuts has created two extra sections, and together, they divide the circle into 6 equal parts. Thus, we only needed 3 cuts, which is exactly what the solution approach predicted for an even number of slices.

Solution Implementation

Python

```python
class Solution:
    def number_of_cuts(self, n: int) -> int:
        # If 'n' is greater than 1 and 'n' is odd, return 'n' itself.
        # The bitwise '&' operator is used to check if 'n' is odd.
        if n > 1 and n & 1:
            return n
        # If 'n' is not greater than 1 or not odd, return 'n' divided by 2.
        # The right shift operator '>>' effectively divides 'n' by 2.
        else:
            return n >> 1
```

Java

```java
class Solution {
    // Method to determine the number of cuts
    public int numberOfCuts(int n) {
        // Check if 'n' is greater than 1 and odd
        // If 'n' is odd and more than 1, return 'n' as the number of cuts
        if (n > 1 && n % 2 == 1) {
            return n;
        } else {
            // If 'n' is even, return half of 'n'
            // The riaht shift operator (>>) divides 'n' by 2
            return n >> 1;
        }
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the number of cuts needed
    int numberOfCuts(int n) {
        // If 'n' is greater than 1 and odd, return 'n'
        if (n > 1 && n % 2 == 1) {
            return n;
        }
        // Otherwise, if 'n' is even, return 'n' divided by 2
        else {
            // Right shift by one position is equivalent to dividing by 2
            return n >> 1;
        }
    }
};
```

TypeScript

```typescript
/**
 * Calculates the number of cuts needed based on the provided number.
 *
 * @param {number} n - The total number of items to make cuts on.
 * @return {number} The number of cuts necessary.
 */
function numberOfCuts(n: number): number {
    // If n is greater than 1 and odd, return n itself
    if (n > 1 && n % 2 !== 0) {
        return n;
    }
    // If n is not greater than 1 or is even, divide n by 2 using bitwise shift and return
    return n >> 1;
}
```

```python
class Solution:
    def number_of_cuts(self, n: int) -> int:
        # If 'n' is greater than 1 and 'n' is odd. return 'n' itself.
        # The bitwise '&' operator is used to check if 'n' is odd.
        if n > 1 and n & 1:
            return n
        # If 'n' is not greater than 1 or not odd, return 'n' divided by 2.
        # The right shift operator '>>' effectively divides 'n' by 2.
        else:
            return n >> 1
```

Time and Space Complexity

Time Complexity

The provided code consists of a single `return` statement that performs a conditional check and either returns `n` directly or performs a bitwise shift to the right (`n >> 1`). Both operations, the conditional check and the bitwise shift, are executed in constant time, which means the time complexity is `O(1)`.

Space Complexity

The space complexity of the function is also `O(1)` because no additional space is used that grows with the input size `n`. The space required for the algorithm is constant, as it only stores a few variables for its operations irrespective of the size of the input.