2927. Distribute Candies Among Children III

Hard Math Combinatorics

Problem Description

The problem presents us with a scenario where we have `n` candies to distribute among 3 children, with a constraint that no child may receive more than `limit` candies. The objective is to calculate the total number of possible distributions that adhere to these rules. Essentially, we're being asked to count the distinct ways to divide the given `n` candies such that each child gets at least 0 candies and at most `limit` candies.

Intuition

The intuition behind the solution is based on combinatorial mathematics, particularly using combinations to calculate the number of ways to perform a task. Since there are no distinctions between candies, the problem is simplified to a partitioning issue: How can `n` identical items (in this case, candies) be divided into three groups with a certain limit?

Initially, we can ignore the limit and consider the number of ways to distribute `n` candies among 3 children. By adding 2 virtual candies (which represent partitions or 'dividers' in our groupings), we can then use combinations to determine the number of ways to place these partitions among the candies such that we end up grouping them into three parts. Mathematically, that's `C(n + 2, 2)`.

However, this initial count includes distributions where children might have more than `limit` candies. To correct for this, we have to exclude these invalid distributions. If any child gets more than `limit` candies, there must be at least `limit + 1` candies in at least one group. We remove these scenarios by calculating how many ways we can distribute the remaining `n - (limit + 1)` candies plus 2 extra for the partitions so that we exclude cases with more than `limit` candies for one child. For each of these possibilities, there are 3 children to which it could happen, which is why we subtract `3 * C(n - limit + 1, 2)`.

The Principle of Inclusion-Exclusion is then applied to account for the possibility that we may have removed too many distributions. Specifically, there could be cases where two children each received more than `limit` candies. We have excluded these cases twice, so we need to add them back in once. Thus, we add `3 * C(n - 2 * limit, 2)`, correcting our over-exclusion.

Our end result combines the initial total count of ways to distribute, the deducted amount for cases where a child has more than `limit` candies, and the corrected count for the overlap where two children might exceed the limit. This gives us the total number of valid distributions following the rules.

Solution Approach

The solution approach is a direct application of the combinatorial method and the Principle of Inclusion-Exclusion. Here's how the algorithm is applied:

1. **Initialization**: It's initially understood that distributing `n` candies among 3 children can be imagined as finding the number of ways to insert two partitions in an arrangement of `n` items. We use the `comb` function from a combinatorial module to calculate combinations.
2. **Virtual Items Addition**: We add 2 virtual items (or virtual candies) to the original `n` candies. This will help us include cases where children can receive zero candies. The virtual items act as dividers, and we are looking to choose 2 spots out of `n + 2` items (real + virtual) for the dividers.
3. **Initial Calculation**: The total number of ways without considering the limit is `C(n + 2, 2)`. This is computed using the `comb` method which calculates combinations. Here `comb(n + 2, 2)` represents the combination formula `n + 2` choose `2`.
4. **Exclude Distributions Exceeding Limit**: Next, we exclude distributions where a child gets more than `limit` candies. If one child gets more than the limit, say `limit + 1`, then we are left with `n - (limit + 1)` candies to distribute among the 3 children (plus 2 virtual candies). This is calculated as `3 * comb(n - limit + 1, 2)`. Since this can occur for any of the 3 children, it's multiplied by 3.
5. **Correct Over-Exclusions**: If two children each receive more than the limit, we will have subtracted too much, as that configuration is excluded in the step above once for each child. To correct this, we add back configurations where two children exceed the limit: `3 * comb(n - 2 * limit, 2)`.
6. **Compiling the Final Answer**: The final answer is a combination of the initial total (`comb(n + 2, 2)`), the first adjustment to exclude invalid distributions (`-3 * comb(n - limit + 1, 2)`), and the inclusion-exclusion correction (`+3 * comb(n - 2 * limit, 2)`).
7. **Interpreting Zero Distribution Case**: If `n` is larger than `3 * limit`, it means that it's impossible to distribute candies without violating the limit constraint for at least one child. Hence, in such a case, the function immediately returns 0 as no valid distribution is possible.

Schema-wise, the function does not use Data Structures other than the internal structure of the combination function it relies on. The algorithm uses the Principle of Inclusion-Exclusion and basic combinatorial patterns to arrive at the solution.

Example Walkthrough

Let us take an example where we have `n = 7` candies to distribute among 3 children, and the `limit` of candies that a single child can receive is 3.

Step 1: Initial Calculation without Limit Ignoring the limit for a moment, we want to find the total number of ways to divide these 7 candies. Including 2 virtual candies as dividers, we have `n + 2 = 9` spots to consider. We need to choose 2 spots for the dividers out of these 9, which is given by the formula `C(n + 2, 2)`. Using the combination formula, this is `C(9, 2) = 36` possible distributions.

Step 2: Exclude Distributions Where a Child Gets More than the Limit Now, we need to exclude the scenarios where a child gets more than 3 candies. To do so, if we give one child 4 candies, we have `7 - 4 = 3` candies left to distribute, including 2 virtual dividers, hence `n - (limit + 1) + 2 = 4` spots. We choose 2 spots for the dividers, which gives us `C(4, 2) = 6`. Since there are 3 children any one of whom could receive more than the limit, we multiply by 3, resulting in `3 * 6 = 18` cases to subtract.

Step 3: Correct Over-Exclusions with Inclusion-Exclusion Principle It's possible that we've double-counted scenarios where two children each have received more than 3 candies. As we subtracted invalid distributions for each child, such scenarios were counted twice. Thus, we need to add back these over-counted distributions. If two children each receive 4 candies, we are left with `7 - 2*(limit + 1) = -1` candies, which doesn't make physical sense and does not need to be added since having a negative number of candies is impossible.

Step 4: Compiling the Final Answer We combined the initial total of ways to distribute (`36`), subtract the first adjustment for invalid distributions (`-18`), and do not have any additional adjustments because the count came to be negative. So the total count of valid distributions, in this case, is `36 - 18 = 18`.

Thus, there are 18 ways to distribute 7 candies among 3 children with each child receiving a maximum of 3 candies.

> Note: If `n > 3 * limit`, we would return 0, as it would be impossible to distribute candies within the set limit. However, in our example, `n = 7` and `limit = 3`, so `n` is not greater than `3 * limit` (which is 9).

Solution Implementation

Python

```python
from math import comb

class Solution:
    def distributeCandies(self, candies: int, limit: int) -> int:
        # Check if the total number of candies is too large to distribute within the limit
        if candies > 3 * limit:
            # If so, return 0 as the distribution is not possible
            return 0

        # Calculate the possible distributions without any constraints
        # This is done by choosing 2 from 'candies + 2', which reoresents
        # distributing candies into 3 slots with restrictions (the Partitions of n)
        possible_distributions = comb(candies + 2, 2)

        # Now we need to subtract the invalid distributions
        # where one child would get more than the limit
        if candies > limit:
            # Subtract the combinations where any one child gets more than the limit
            possible_distributions -= 3 * comb(candies - limit + 1, 2)

        # Further adjust the count if one child gets more than double the limit
        if candies - 2 >= 2 * limit:
            # Add back the combinations that we subtracted
            # twice due to overlap in our previous calculation
            possible_distributions += 3 * comb(candies - 2 * limit, 2)

        # Return the final number of possible distributions
        return possible_distributions
```

Java

```java
class Solution {

    // Method to distribute candies in n number of ways without exceeding the limit per color
    public long distributeCandies(int candies, int limit) {

        // If the number of candies is more than 3 times the limit, it's not possible to distribute
        if (candies > 3 * limit) {
            return 0;
        }

        // Calculate initial combination for distribution
        long distributionCount = calculateCombination(candies + 2);

        // If the number of candies exceeds the limit, reduce impossible combinations
        if (candies > limit) {
            distributionCount -= 3 * calculateCombination(candies - limit + 1);
        }

        // If twice the limit is less than the candies minus two,
        // add combinations where the number of candies for any color
        // would not exceed the limit
        if (candies - 2 >= 2 * limit) {
            distributionCount += 3 * calculateCombination(candies - 2 * limit);
        }

        // Return the final count of distributions
        return distributionCount;
    }

    // Helper method to calculate the number of combinations selecting 2 from n items (nC2)
    private long calculateCombination(int n) {
        // Using the combination formula nC2 = n! / (2!(n-2)!)
        // Which simplifies to n*(n-1)/2
        return 1L * n * (n - 1) / 2;
    }
}
```

C++

```cpp
class Solution {
public:
    // Helper function to calculate the number of unique pairs (combinations of 2) from 'n' items
    long long combinationsOfTwo(int n) {
        return 1LL * n * (n - 1) / 2;
    }

    // Function to calculate how many distributions of candies are possible given constraints
    long long distributeCandies(int candies, int limit) {
        // If there are more than three times the limit of candies, no distributions are possible
        if (candies > 3 * limit) {
            return 0;
        }

        // Start with the total combinations of 'candies + 2' taken 2 at a time
        long long distributionCount = combinationsOfTwo(candies + 2);

        // If there are more candies than the limit, we need to subtract impossible distributions
        if (candies > limit) {
            distributionCount -= 3 * combinationsOfTwo(candies - limit + 1);
        }

        // If there are at least twice the limit after distributing two candies.
        // add back the combinations previously subtracted that are now possible
        if (candies - 2 >= 2 * limit) {
            distributionCount += 3 * combinationsOfTwo(candies - 2 * limit);
        }

        // Return the final count of possible candy distributions
        return distributionCount;
    }
};
```

TypeScript

```typescript
function distributeCandies(candies: number, limit: number): number {
    // Function to calculate the number of ways to choose 2 from n (n choose 2)
    const choose2 = (num: number): number => (num * (num - 1)) / 2;

    // If the number of candies is more than triple the limit, no valid distribution possible
    if (candies > 3 * limit) {
        return 0;
    }

    // Start with the number of ways to choose 2 candies from (candies + 2)
    let answer = choose2(candies + 2);

    // If there are more candies than the limit, we need to subtract invalid combinations
    if (candies > limit) {
        answer -= 3 * choose2(candies - limit + 1);
    }

    // If the number of remaining candies (after giving away limit candies twice) is still above the limit,
    // Add back combinations that were previously subtracted
    if (candies - 2 >= 2 * limit) {
        answer += 3 * choose2(candies - 2 * limit);
    }

    // Return the final answer
    return answer;
}
```

```python
from math import comb

class Solution:
    def distributeCandies(self, candies: int, limit: int) -> int:
        # Check if the total number of candies is too large to distribute within the limit
        if candies > 3 * limit:
            # If so, return 0 as the distribution is not possible
            return 0

        # Calculate the possible distributions without any constraints
        # This is done by choosing 2 from 'candies + 2', which reoresents
        # distributing candies into 3 slots with restrictions (the Partitions of n)
        possible_distributions = comb(candies + 2, 2)

        # Now we need to subtract the invalid distributions
        # where one child would get more than the limit
        if candies > limit:
            # Subtract the combinations where any one child gets more than the limit
            possible_distributions -= 3 * comb(candies - limit + 1, 2)

        # Further adjust the count if one child gets more than double the limit
        if candies - 2 >= 2 * limit:
            # Add back the combinations that we subtracted
            # twice due to overlap in our previous calculation
            possible_distributions += 3 * comb(candies - 2 * limit, 2)

        # Return the final number of possible distributions
        return possible_distributions
```

Time and Space Complexity

The time complexity of the function `distributeCandies` is `O(1)` because all operations consist of a fixed number of calculations that do not depend on the size of the input `n` or `limit`. The function performs basic arithmetic operations and calls the `comb` function a constant number of times (at most 6 times). The `comb` function's complexity is constant in this context because it is likely implemented to work in constant time for the inputs it receives, typically using precomputed values or efficient formulas for combination calculations.

The space complexity of the function is also `O(1)`. The reason is that the function only uses a fixed amount of space for the variables `ans`, `n`, and `limit`, and does not allocate any additional space that grows with the size of the input. It uses a constant amount of space for temporary variables used in the combination function and to store the result regardless of the input size.