2929. Distribute Candies Among Children II

Medium Math Combinatorics Enumeration

Problem Description

You are given a task to distribute `n` candies among `3` children. The goal is to figure out the total number of ways to do this. However, there's an additional constraint that must be considered: no child is allowed to receive more than `limit` candies. The problem asks us to calculate all the possible distributions that respect this constraint.

At its core, this problem is about combinations and understanding the limits of distribution. It's similar to partitioning objects into groups with an upper bound on the group size. Imagining the candies as items to be placed in containers (one for each child), where a container can hold up to a certain number of items (the `limit`), might help to visualize the problem.

Given these conditions, your task is to figure out how to count all permissible arrangements where each child can only have a certain maximum number of candies. This simple scenario is complicated by the upper cap of candies that a single child can receive.

Intuition

The intuition behind the solution relies on principles from combinatorial mathematics and the principle of inclusion-exclusion.

We start with the notion that distributing `n` candies among `3` children is similar to deciding how to place `n` indistinguishable balls into `3` distinguishable bins. Here, the candies are the balls, and each child represents a bin.

Initially, without considering the `limit`, we can calculate the number of ways by adding 2 virtual balls (or placeholders for the partitions) to the `n` candies. This transforms the problem into one of placing partitions between balls, which is a common combinatorial approach. The placements of these partitions split the `n` candies into 3 groups, representing the share for each child. The formula for this would be combinations of `n + 2` taken `2` at a time (`C(n + 2, 2)`).

However, this method counts distributions where a child might get more than the `limit`. Thus, we must exclude these possibilities. For each child, if they get more than the `limit`, the possible distributions for the remaining candies, including the virtual ones, diminish. Here we subtract the combinations where one of the children has exceeded the limit (`3 * C(n - limit + 1, 2)`).

The inclusion-exclusion principle comes into play when we've subtracted too much; some distributions have been excluded twice because they have two children receiving more than the 'limit'. We need to add these back one time to correct the count (`3 * C(n - 2 * limit, 2)`).

That's the overall approach, which uses combinatorial mathematics to find the total number of distributions while making adjustments based on the `limit` to exclude or re-include certain possibilities as per the problem's conditions.

Solution Approach

The solution approach to this problem uses a Python class `Solution` with a method `distributeCandies` that takes `n` and `limit` as its parameters and returns the total number of ways to distribute the candies among `3` children according to the given constraints.

- **Early Termination Condition**: The implementation checks if `n` is greater than `3 * limit`; if this is true, it means that there is no possible way to distribute the candies without breaking the maximum limit for at least one child. Therefore, the method returns 0 as the number of ways.
- **Initial Combination Computation**: If the early termination condition is not met, the method then computes the initial number of possible distributions ignoring the limit. This is done using the combinatorial formula `C(n + 2, 2)`, which translates to the number of ways to choose `2` partitions from `n + 2` options (initial number of candies plus 2 virtual balls).
- **Exclusion of Over-limit Distributions**: Next, if there are more candies than `limit`, the method calculates and subtracts from the total the number of distributions where at least one child gets more than `limit` candies. This is done by computing `3 * C(n - limit + 1, 2)` —the number of possible distributions for the excess candies while considering all three children.
- **Inclusion-Exclusion Principle**: Lastly, if `n - 2` is greater than or equal to `2 * limit`, the distributions where two boxes have exceeded the limit have been subtracted twice (once for each child). To correct this, the method adds back the number of possibilities of over-limit distributions for two children, which is computed by `3 * C(n - 2 * limit, 2)`.
- **Return the Corrected Total**: The resulting number of distributions, after including and excluding the appropriate counts, represents the total number of ways candies can be distributed according to the rules.

The combinatorial calculations are likely made using a function or formula that computes combinations (`comb`). The combination formula for `C(n, k)` is essentially `n! / (k! * (n - k)!)`, where `!` represents the factorial operation.

In summary, the implementation uses combinatorial mathematics to count the initial number of distributions and then applies the principle of inclusion-exclusion to adjust the count, adhering to the constraints of the distribution limit for each child. The solution showcases a thoughtful use of combinatorial patterns to solve a constrained distribution problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the case where we have `n = 5` candies to distribute among `3` children, with a `limit = 2` candies per child.

We'll follow the steps outlined in the solution approach:

1. **Early Termination Condition**: First, we need to check whether distributing `5` candies is possible without exceeding the limit of `2` candies per child. Since `n = 5` is not greater than `3 * limit` (which is `3 * 2 = 6`), we do not terminate early. There is at least one way to distribute the candies without breaking the maximum limit for any child.
2. **Initial Combination Computation**: Without considering the limit, we would calculate the number of ways to distribute the `5` candies using the combinatorial formula `C(n + 2, 2)`. Here, `n + 2` equals `7` (which includes the 5 candies plus 2 virtual balls representing partitions). So we need to compute the number of ways to choose `2` partitions from `7` options, which is `C(7, 2)`. This is equal to `7! / (2! * (7 - 2)!) = 21` ways.
3. **Exclusion of Over-limit Distributions**: There are more candies than the `limit` per child, so we need to subtract the combinations where one child gets more than `2` candies. We compute `3 * C(5 - 2 * 1, 2)`, which simplifies to `3 * C(4, 2)`. This equals `3 * (4! / (2! * (4 - 2)!)) = 3 * 6 = 18` ways that must be excluded (where 3 represents the number of children).
4. **Inclusion-Exclusion Principle**: Since `5 - 2` is not greater or equal to `2 * limit` (which would be `4`), this step doesn't apply. There are no possibilities of over-limit distributions for two children to add back.
5. **Return the Corrected Total**: The total number of permissible distributions, after excluding the over-limit scenarios, is `21 - 18 = 3` ways.

Thus, if we have `5` candies and a limit of `2` candies per child, there are `3` ways to distribute the candies without any child receiving more than `2` candies. The three distributions meeting the criteria would be as follows: (2, 2, 1), (2, 1, 2), and (1, 2, 2).

This example illustrates how the proposed solution approach applies the principle of combinatorial mathematics and the principle of inclusion-exclusion to solve the problem with the given constraints.

Solution Implementation

Python

```python
from math import comb

class Solution:
    def distributeCandies(self, candidates: int, limit: int) -> int:
        # If candidates are more than three times the limit, no distribution is possible.
        if candidates > 3 * limit:
            return 0

        # Calculate the total possible distributions without any restrictions
        # This is the number of combinations to distribute `candidates` into 3 boxes
        # (ans is short for answer and represents the total number of distributions)
        total_distributions = comb(candidates + 2, 2)

        # Subtract the distributions that exceed the limit in any box
        # This happens when the number of candidates exceeds the limit
        if candidates > limit:
            total_distributions -= 3 * comb(candidates - limit + 1, 2)

        # Add back the distributions that were subtracted more than once
        # This correction is needed when candidates are twice over the limit,
        # as those would be subtracted twice in the previous step
        if candidates - 2 >= 2 * limit:
            total_distributions += 3 * comb(candidates - 2 * limit, 2)

        # Return the final count of valid distributions
        return total_distributions
```

Java

```java
class Solution {

    // This method is designed to distribute candies among three children with a certain limit
    public long distributeCandies(int candies, int limit) {

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

        // Calculate the basic number of distributions without limits using combinatorics
        long distributions = combinationOfTwo(candies + 2);

        // Adjust the distribution count by accounting for the limit
        if (candies > limit) {
            distributions -= 3 * combinationOfTwo(candies - limit + 1);
        }

        // Further adjust the distribution if needed when candies are twice over the limit
        if (candies - 2 >= 2 * limit) {
            distributions += 3 * combinationOfTwo(candies - 2 * limit);
        }

        // Return the total number of valid distributions
        return distributions;
    }

    // Helper method to calculate combinations, choosing 2 from n (nC2)
    private long combinationOfTwo(int n) {
        // Use long to avoid integer overflow for large n values
        return 1L * n * (n - 1) / 2;
    }
}
```

C++

```cpp
class Solution {
public:
    long long distributeCandies(int candies, int limit) {
        // Define a lambda function to calculate combinations of 2 from a given number
        auto calculateComb2 = [](int number) -> long long {
            return static_cast<long long>(number) * (number - 1) / 2;
        };

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

        // Calculate the base number of distributions for n + 2
        long long distributionCount = calculateComb2(candies + 2);

        // If there are more candies than the limit, subtract invalid distributions
        if (candies > limit) {
            distributionCount -= 3 * calculateComb2(candies - limit + 1);
        }

        // Add back any distributions that were subtracted twice
        if (candies - 2 >= 2 * limit) {
            distributionCount += 3 * calculateComb2(candies - 2 * limit);
        }

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

TypeScript

```typescript
// Function to calculate the number of combinations to distribute candies
// n represents the number of candies
// limit represents the maximum number of candies per person
function distributeCandies(candies: number, limit: number): number {
    // Helper function to compute combinations of 2 from n
    const combinationsOfTwo = (n: number) => (n * (n - 1)) / 2;

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

    // Calculate basic combination count for candies + 2
    let answer = combinationsOfTwo(candies + 2);

    // If there are more candies than the limit, adjust the number
    // of combinations by subtracting impossible distributions
    if (candies > limit) {
        answer -= 3 * combinationsOfTwo(candies - limit + 1);
    }

    // If the number of candies minus 2 is at least double the limit,
    // adjust the number of combinations by adding back some distributions
    if (candies - 2 >= 2 * limit) {
        answer += 3 * combinationsOfTwo(candies - 2 * limit);
    }

    // Return the final count of valid candy distributions
    return answer;
}
```

```python
from math import comb

class Solution:
    def distributeCandies(self, candidates: int, limit: int) -> int:
        # If candidates are more than three times the limit, no distribution is possible.
        if candidates > 3 * limit:
            return 0

        # Calculate the total possible distributions without any restrictions
        # This is the number of combinations to distribute `candidates` into 3 boxes
        # (ans is short for answer and represents the total number of distributions)
        total_distributions = comb(candidates + 2, 2)

        # Subtract the distributions that exceed the limit in any box
        # This happens when the number of candidates exceeds the limit
        if candidates > limit:
            total_distributions -= 3 * comb(candidates - limit + 1, 2)

        # Add back the distributions that were subtracted more than once
        # This correction is needed when candidates are twice over the limit,
        # as those would be subtracted twice in the previous step
        if candidates - 2 >= 2 * limit:
            total_distributions += 3 * comb(candidates - 2 * limit, 2)

        # Return the final count of valid distributions
        return total_distributions
```

Time and Space Complexity

The time complexity of the code is `O(1)`. This is because the operations consist of a fixed number of mathematical operations: addition, multiplication, and function `comb`, which calculates the binomial coefficient. The `comb` function internally likely uses a direct formula for calculation, and thus it takes a constant time regardless of the input size.

The space complexity of the code is also `O(1)`. The reason for this is that the amount of memory used does not scale with the size of the input `n`; only a fixed number of variables (`ans`) and constants are in use, and the space required for the computation of the `comb` function is also constant.