1103. Distribute Candies to People

Easy Math Simulation

Leetcode Link

Problem Description

In this problem, we have a certain number of `candies` that need to be distributed to `num_people` people arranged in a row. The distribution starts with the first person receiving one candy, the second person receiving two candies, and so on, increasing the count of candies by one for each subsequent person until the `n`th person receives `n` candies. After reaching the last person, the distribution continues from the first person again, but this time each person gets one more candy than the previous cycle (so the first person now gets `n+1` candies, the second gets `n+2`, and so on). This process repeats until we run out of candies. If there are not enough candies to give the next person in the sequence their "full" amount, they receive the remaining candies, and the distribution ends.

The goal is to return an array of length `num_people`, with each element representing the total number of candies that each person receives at the end of the distribution process.

Intuition

To solve this problem, we want to simulate the described candy distribution process. We keep handing out candies until we have none left. Each person gets a certain number of candies based on the round of distribution we are in. In the first round, person 1 gets 1 candy, person 2 gets 2 candies, and so on. Once we reach `num_people`, we wrap around and start from person 1 again, increasing the amount of candy given out by `num_people` each round.

The solution involves iterating over the people in a loop and incrementing the number of candies each person gets by the distribution rule given. We maintain a counter `i` to keep track of how many candies have been given out so far, and a list `ans` to store the total candies for each person.

With each person's turn, we give out the number of candies equal to the counter `i + 1`, but if we have fewer candies left than `i + 1`, we give out all the remaining candies. After that, we update the total number of candies left by subtracting the number given out. If the candies finish during someone's turn, we stop the distribution and return our `ans` list to show the final distribution of candies.

The intuition is to replicate the physical process of handing out the candies in a loop, ensuring that conditions such as running out of candies are properly handled.

Solution Approach

The solution to this problem uses a simple iterative approach as our algorithm. Here are the steps and the reasoning in detail:

1. **Initialize the Answer List:** We start by initializing an array `ans` of length `num_people` with all elements set to 0. This array will be used to keep track of the number of candies each person receives.

2. **Starting the Distribution:** We need to keep track of two things - the index of the person to whom we're currently giving candies, and the number of candies we're currently handing out. We start the distribution by setting a counter `i` to 0, which will increase with each iteration to represent the amount of candy to give.

3. **Iterative Distribution:** We use a `while` loop to continue distributing candies until we run out (`candies > 0`). During each iteration of the loop:

    - Compute `i % num_people` to find the index of the current person. This ensures that after the last person, we start again from the first person.
    - Determine the number of candies to give to the current person. We use `min(candies, i + 1)` to decide this amount because we either give `i + 1` candies or the remaining candies if we have less than `i + 1`.
    - Subtract the number of distributed candies from `candies`.
    - Increment `i` by 1 to update the count for the next iteration.
4. **Updating Answer List:** In each iteration, update `ans[i % num_people]` with the number of candies distributed in that iteration.

5. **Handling Remaining Candies:** If we deplete our supply of `candies`, we give out the remaining candies to the last person. This is built into the allocation step with `min(candies, i + 1)`.

6. **Returning the Final Distribution:** Once the loop ends (no more candies are left), we exit the loop. The array `ans` now contains the total number of candies received by each person, which we return as the final answer.

This problem does not require any complex data structures or patterns. The concept is straightforward and only uses basic array manipulation to achieve the goal. It focuses on handling the loop correctly and ensuring that the distribution of candies is done according to the specified rules.

```python
class Solution:
    def distributeCandies(self, candies: int, num_people: int) -> List[int]:
        ans = [0] * num_people  # Initialize the answer list
        i = 0                   # Counter for the distribution process
        # Distribute candies until we run out
        while candies:
            # Give out min(candies, i + 1) candies to the (i % num_people)th person
            ans[i % num_people] += min(candies, i + 1)
            candies -= min(candies, i + 1) # Subtract the candies given from the total
            i += 1  # Move to the next person

        return ans  # Return the final distribution
```

The solution makes effective use of modulo operation to cycle through the indices repeatedly while the `while` loop condition ensures that the distribution halts at the right time.

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose we have `candies = 7` and `num_people = 4`.

We want to distribute these candies across 4 people as described. Let's walk through the process using the provided algorithm.

1. **Initialize the Answer List:**
    - `ans = [0, 0, 0, 0]`
2. **Starting the Distribution:**
    - `candies = 7`
    - Distribution counter `i = 0`
3. **Iterative Distribution:**
    1. During the first iteration (`i = 0`):
        - Current person index: `0 % 4 = 0` (first person)
        - Candies to give out: `min(7, 0 + 1) = 1`
        - Remaining candies: `7 - 1 = 6`
        - Updated `ans` list: `[1, 0, 0, 0]`
        - Increment `i` to `1`
    2. In the second iteration (`i = 1`):
        - Current person index: `1 % 4 = 1` (second person)
        - Candies to give out: `min(6, 1 + 1) = 2`
        - Remaining candies: `6 - 2 = 4`
        - Updated `ans` list: `[1, 2, 0, 0]`
        - Increment `i` to `2`
    3. In the third iteration (`i = 2`):
        - Current person index: `2 % 4 = 2` (third person)
        - Candies to give out: `min(4, 2 + 1) = 3`
        - Remaining candies: `4 - 3 = 1`
        - Updated `ans` list: `[1, 2, 3, 0]`
        - Increment `i` to `3`
    4. In the fourth iteration (`i = 3`):
        - Current person index: `3 % 4 = 3` (fourth person)
        - Candies to give out: `min(1, 3 + 1) = 1`
        - Remaining candies: `1 - 1 = 0` (no more candies)
        - Updated `ans` list: `[1, 2, 3, 1]`
        - Candies are now depleted, we stop the distribution.
4. **Return the Final Distribution:**
    - The final `ans` list is `[1, 2, 3, 1]`.

Each element in the `ans` list represents the total number of candies each person receives after the distribution is done. The algorithm successfully mimics the handing out of candies until there are no more left, while following the rules set out in the problem description.

Python Solution

```python
from typing import List

class Solution:
    def distributeCandies(self, candies: int, num_people: int) -> List[int]:
        # Initialize a list to hold the number of candies each person will receive
        distribution = [0] * num_people
        # Initialize an index variable to distribute candies to the people in order
        index = 0

        # Continue distribution until there are no more candies left
        while candies > 0:
            # Calculate the number of candies to give: either 1 more than the current index
            # or the remaining candies if fewer than that number remain
            give = min(candies, index + 1)
            # Distribute the candies to the current person
            distribution[index % num_people] += give
            # Subtract the number of candies given from the remaining total
            candies -= give
            # Move to the next person for the next round of distribution
            index += 1

        # Return the final distribution
        return distribution

```

Java Solution

```java
class Solution {
    public int[] distributeCandies(int candies, int numPeople) {
        // Initialize the answer array with the size equal to numPeople.
        // All elements are initialized to 0.
        int[] distribution = new int[numPeople];

        // Initialize the index for the current person
        // and the amount to give out to the current person
        int index = 0;
        int currentCandyAmount = 1;

        // Use a loop to distribute the candies until all candies are distributed
        while (candies > 0) {
            // Calculate the index for the current distribution round
            // It cycles back to 0 when it reaches numPeople
            int personIndex = index % numPeople;

            // Determine the number of candies to give out
            // It is the minimum of either the remaining candies or the current amount
            int candiesToGive = Math.min(candies, currentCandyAmount);

            // Update the candies count for the current person
            distribution[personIndex] += candiesToGive;

            // Subtract the candies given out from the total count of remaining candies
            candies -= candiesToGive;

            // Move to the next person and increment the candy amount
            index++;
            currentCandyAmount++;
        }

        // Return the distribution result
        return distribution;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::min function

class Solution {
public:
    /**
     * Distributes candies among people in a loop.
     *
     * @param candies Number of candies to distribute.
     * @param num_people Number of people to distribute the candies to.
     * @return A vector<int> containing the distribution of candies.
     */
    vector<int> distributeCandies(int candies, int num_people) {
        vector<int> distribution(num_people, 0); // Create a vector with num_people elements, all initialized to 0
        int i = 0; // Initialize a counter to track the number of candies given

        // Continue distributing candies until none are left
        while (candies > 0) {
            // Calculate the index of the current person and the amount of candies to give
            int index = i % num_people;
            int give = std::min(candies, i + 1); // The number of candies to give is the lesser of the remaining candies and the curr

            distribution[index] += give; // Distribute the candies to the current person
            candies -= give; // Decrease the total candy count

            ++i; // Move to the next candy count
        }

        return distribution; // Return the final distribution
    }
};

```

Typescript Solution

```typescript
// Function to distribute candies among people in a way that the ith allocation
// increases by 1 candy
function distributeCandies(candies: number, numPeople: number): number[] {
    // Initialize an answer array to hold the number of candies for each person,
    // starting with zero candies for each person
    const distribution: number[] = new Array(numPeople).fill(0);

    // Variable to track the current distribution round
    let currentDistribution = 0;

    // Continue distributing candies until none are left
    while (candies > 0) {
        // Calculate the current person's index by using modulo with numPeople.
        // This ensures we loop over the array repeatedly
        const currentIndex = currentDistribution % numPeople;

        // Determine the number of candies to give in this round. It is the minimum
        // of the remaining candies and the current distribution amount (1-indexed)
        const candiesToGive = Math.min(candies, currentDistribution + 1);

        // Update the distribution array for the current person
        distribution[currentIndex] += candiesToGive;

        // Subtract the given candies from the total remaining candies
        candies -= candiesToGive;

        // Move on to the next round of distribution
        currentDistribution++;
    }

    // Return the final distribution of candies
    return distribution;
}

```

Time and Space Complexity

The time complexity of the given code can be determined by the while loop, which continues until all `candies` are distributed. In each iteration of the loop, `i` is incremented by `1`, and the amount of candies distributed is also incremented by `1` until all candies are exhausted. This forms an arithmetic sequence from `1` to `n` where `n` is the turn where the candies run out. The total number of candies distributed by this sequence can be represented by the sum of the first `n` natural numbers formula `n*(n+1)/2`. So the time complexity is governed by the smallest `n` such that `n*(n+1)/2 >= candies`. Therefore, the time complexity is O(sqrt(candies)) because we need to find an `n` such that `n^2` is asymptotically equal to the total number of `candies`.

The space complexity of the code is determined by the list `ans` that has a size equal to `num_people`. Since the size of this list does not change and does not depend on the number of candies, the space complexity is O(num_people), which is the space required to store the final distribution of the candies among the people.