<https://leetcode.com/problems/distribute-candies-to-people/description/>

We distribute some number of candies, to a row of **n = num\_people** people in the following way:

We then give 1 candy to the first person, 2 candies to the second person, and so on until we give n candies to the last person.

Then, we go back to the start of the row, giving n + 1 candies to the first person, n + 2 candies to the second person, and so on until we give 2 \* n candies to the last person.

This process repeats (with us giving one more candy each time, and moving to the start of the row after we reach the end) until we run out of candies.  The last person will receive all of our remaining candies (not necessarily one more than the previous gift).

Return an array (of length num\_people and sum candies) that represents the final distribution of candies.

**Example 1:**

**Input:** candies = 7, num\_people = 4

**Output:** [1,2,3,1]

**Explanation:**

On the first turn, ans[0] += 1, and the array is [1,0,0,0].

On the second turn, ans[1] += 2, and the array is [1,2,0,0].

On the third turn, ans[2] += 3, and the array is [1,2,3,0].

On the fourth turn, ans[3] += 1 (because there is only one candy left), and the final array is [1,2,3,1].

**Example 2:**

**Input:** candies = 10, num\_people = 3

**Output:** [5,2,3]

**Explanation:**

On the first turn, ans[0] += 1, and the array is [1,0,0].

On the second turn, ans[1] += 2, and the array is [1,2,0].

On the third turn, ans[2] += 3, and the array is [1,2,3].

On the fourth turn, ans[0] += 4, and the final array is [5,2,3].

**Constraints:**

1 <= candies <= 10^9

1 <= num\_people <= 1000

**Attempt 1: 2025-09-07**

**Solution 1: Math (30 min)**

class Solution {

    public int[] distributeCandies(int candies, int num\_people) {

        int[] result = new int[num\_people];

        int cur = 1;

        int index = 0;

        while(candies > 0) {

            // Check if we have enough candies to distribute next

            if(candies >= cur) {

                result[index] += cur;

                candies -= cur;

                index = (index + 1) % num\_people;

                cur++;

            // If we don't have enough candies to distribute

            } else {

                // Distribute whatever is remaining

                result[index] += candies;

                // Set to 0 so it exits the while loop

                candies = 0;

            }

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

I'll provide a Java solution for LeetCode 1103. "Distribute Candies to People", along with an explanation and complexity analysis.

**Problem Description**

We distribute some number of candies to a row of num\_people in the following way:

We give 1 candy to the first person, 2 candies to the second person, and so on until we give n candies to the last person.

Then, we go back to the start of the row, giving n+1 candies to the first person, n+2 to the second, and so on.

This repeats until we run out of candies.

Return an array of length num\_people that represents the final distribution of candies.

**Java Solution**

class Solution {

public int[] distributeCandies(int candies, int num\_people) {

int[] result = new int[num\_people];

int index = 0;

int currentCandy = 1;

while (candies > 0) {

// Give min(currentCandy, candies) to the current person

int give = Math.min(currentCandy, candies);

result[index] += give;

candies -= give;

// Move to next person and increment candy count

index = (index + 1) % num\_people;

currentCandy++;

}

return result;

}

}

**Explanation**

**Initialization**: We create an array result of length num\_people to store the candies each person receives.

**Distribution Loop**:

We start with currentCandy = 1 and index = 0 (first person).

In each iteration, we give min(currentCandy, candies) to the current person. This ensures we don't give more candies than we have left.

We subtract the given candies from the total candies.

We move to the next person (using modulo arithmetic to wrap around to the start after the last person) and increment currentCandy for the next distribution.

**Termination**: The loop continues until we run out of candies.

**Example Walkthrough**

Let's take candies = 7, num\_people = 3:

First distribution:

Person0: give 1 → result[0]=1, candies=6, currentCandy=2

Person1: give 2 → result[1]=2, candies=4, currentCandy=3

Person2: give 3 → result[2]=3, candies=1, currentCandy=4

Next round:

Person0: give min(4,1)=1 → result[0]=2, candies=0

Final result: [2, 2, 3]

**Complexity Analysis**

**Time Complexity**: O(sqrt(candies)) or O(num\_people \* rounds). The number of rounds is roughly sqrt(2 \* candies) because the total candies distributed after n rounds is n(n+1)/2.

**Space Complexity**: O(num\_people) for the result array.

This solution is efficient and straightforward, simulating the distribution process until all candies are gone.

**How about my solution ?**

Yes, the solution you provided is also correct and is a common approach to solve this problem. It uses a nested loop structure: an outer while loop that continues until all candies are distributed, and an inner for loop that iterates over each person in the row. Let me break it down and compare it with the previous solution.

**Your Provided Solution:**

class Solution {

public int[] distributeCandies(int candies, int num\_people) {

int[] arr = new int[num\_people];

int currCandyCount = 1;

while(candies > 0) {

for(int i = 0; i < num\_people; i++) {

if(candies >= currCandyCount) {

arr[i] += currCandyCount;

candies -= currCandyCount;

currCandyCount++;

} else {

arr[i] += candies;

candies = 0;

}

}

}

return arr;

}

}

**How It Works:**

**Initialization**:

arr is an array of length num\_people to store the candies each person gets.

currCandyCount starts at 1, representing the number of candies to be given to the next person.

**Outer while loop**:

Continues as long as there are candies left (candies > 0).

**Inner for loop**:

Iterates over each person (index i from 0 to num\_people-1).

For each person:

If there are enough candies to give currCandyCount, then:

Give currCandyCount to arr[i].

Subtract currCandyCount from candies.

Increment currCandyCount for the next person.

Else (if there aren't enough candies for the full currCandyCount):

Give all remaining candies to arr[i].

Set candies to 0 to break out of the loops.

**Termination**:

The inner loop runs for each person in the row, and the outer loop continues until all candies are distributed. The inner loop might run multiple times (multiple rounds) until candies becomes 0.

**Example Walkthrough (candies=7, num\_people=3):**

**First round (inner loop)**:

i=0: candies=7 >= currCandyCount=1 → give 1 to arr[0] → arr[0]=1, candies=6, currCandyCount=2.

i=1: candies=6 >=2 → give 2 to arr[1] → arr[1]=2, candies=4, currCandyCount=3.

i=2: candies=4 >=3 → give 3 to arr[2] → arr[2]=3, candies=1, currCandyCount=4.

**Second round (inner loop)**:

i=0: candies=1 < currCandyCount=4 → give 1 to arr[0] → arr[0]=2, candies=0.

Break out (candies=0).

Result: [2, 2, 3]

**Comparison with the Previous Solution:**

**Similarities**:

Both solutions simulate the distribution process until no candies are left.

Both use an array to store the result and update it incrementally.

Both have the same time and space complexity.

**Differences**:

The first solution uses a single loop with modulo indexing to handle the circular distribution.

The second solution uses a nested loop: the outer while loop and inner for loop to iterate over each person in each round.

The first solution might be slightly more efficient in terms of code structure (avoiding nested loops) but both are acceptable.

**Complexity Analysis:**

**Time Complexity**: O(sqrt(candies)) or O(num\_people \* rounds). The number of rounds (outer loop iterations) is roughly the number of times we complete a full cycle. Since the total candies distributed after k full rounds is k \* (k+1)/2 \* num\_people (approximately), the number of rounds is about O(sqrt(candies)).

**Space Complexity**: O(num\_people) for the result array.

**Which One is Better?**

Both solutions are correct and efficient. The first solution (using modulo) might be considered more elegant because it avoids nested loops. However, the second solution (with nested loops) is straightforward and easy to understand.

You can choose either based on your preference. The nested loop solution is perfectly acceptable and clear.

Let me know if you need further clarification!

**Refer to**

[L2591.Distribute Money to Maximum Children (Ref.L1103,L1716,L2305)](note://WEB8c809777625c21289686266b9573d258)